VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

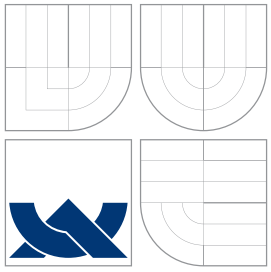
LIDSKÉ ROZHRANÍ K AUTOMATOVÝM KNIHOVNÁM NÁSTROJE MONA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

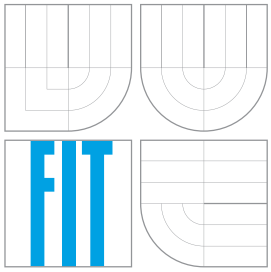
AUTOR PRÁCE
AUTHOR

RADEK PYŠNÝ

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

LIDSKÉ ROZHRANÍ K AUTOMATOVÝM KNIHOVNÁM NÁSTROJE MONA

HUMAN INTERFACE TO AUTOMATA LIBRARIES OF MONA TOOL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

RADEK PYŠNÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Mgr. ADAM ROGALEWICZ, Ph.D.

BRNO 2011

Abstrakt

Konečné stromové automaty jsou formalismem používaným v mnoha různých oblastech informatiky, mimo jiné v oblasti formální verifikace. V současné době existuje několik nástrojů pro práci s konečnými stromovými automaty, avšak knihovny nástroje MONA jsou pro tyto účely nejlepší. Právě konečné stromové automaty jsou častým nástrojem pro formální verifikaci počítačových systémů, které pracují s dynamickými datovými strukturami. Způsob, jakým je realizováno zadávání konečných stromových automatů pro knihovny nástroje MONA, je pro člověka značně náročný, protože je nutné funkci přechodu konečného stromového automatu zadat ve formě několika multiterminálních binárních rozhodovacích diagramů. Cílem této diplomové práce je navrhnout a implementovat nástroj pro převod konečných stromových automatů zapsaných výčtem pravidel do interního formátu nástroje MONA.

Abstract

Finite tree automata is formalism used in many different areas of computer science, among others in the area of formal verification. Nowdays there are few tools used for handling of finite tree automata, however libraries of MONA tool are the best choice. The finite tree automata are a frequent tool for formal verification of computer systems which work with dynamic data structures. The input format of finite tree automata for libraries of MONA tool is very difficult for humans because it is necessary to enter the move function of the finite tree automaton in a form of several multiterminal binary decision diagrams. The aim of this thesis is to design and implement tool to convert the finite set of move rules into internal format of the MONA tool.

Klíčová slova

Formální verifikace, stromové automaty, binární rozhodovací diagramy, rozhraní.

Keywords

Formal verification, tree automata, binary decision diagrams, interface.

Citace

Pyšný, R. *Lidské rozhraní k automatovým knihovnám nástroje MONA*. Brno: FIT VUT v Brně, 2011. Diplomová práce.

Lidské rozhraní k automatovým knihovnám nástroje MONA

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Mgr. Adama Rogalewicze, Ph.D.

.....
Radek Pyšný
25. května 2011

Poděkování

Rád bych touto formou poděkoval vedoucímu práce, Mgr. Adamu Rogalewiczovi, Ph.D., za odborný přínos, ochotu ke konzultacím a cenné rady. Dále bych rád poděkoval své rodině, přátelům a spolužákům za jejich podporu a pomoc při studiu.

Obsah

Kapitola 1

Úvod

Počítačové systémy se staly nedílnou součástí každodenního života každého z nás. Obzvláště v posledních několika letech si můžeme všimnout, jak se v mnoha oblastech stáváme závislými na technických vymoženostech dnešní doby. Často se až nesmyslně spoléháme na různá technická zařízení a programy, které je ovládají. Předpokládáme, že jsou bezchybná, avšak naše předpoklady jsou často mylné. Selhání jednoho jediného takového systému může způsobit nevyčíslitelné finanční ztráty, ekologickou havárii – jednu takovou máme stále v živé paměti z dubna loňského roku – nebo dokonce ztráty na lidských životech. Dnešní počítačové systémy se vyvíjejí skutečně velkým tempem, jejich složitost stále roste a je téměř nad lidské síly udržet tyto systémy v bezproblémovém chodu. Jednou z možností, které jsou schopné zaručit spolehlivost počítačových systémů, je věnovat úsilí oblasti *formální verifikace*. Metody formální verifikace sice nejsou schopny plně nahradit testování, avšak jsou schopny matematicky dokázat, zda systém odpovídá zadaným specifikacím. Do oblasti formální verifikace je v současné době investováno velké množství peněz a spousta lidského úsilí, což pomalu ale jistě přináší své plody. O některých společnostech, jako např. Intel, Microsoft, NASA nebo Airbus, je již obecně známo, že se oblasti formální verifikace věnují celá jejich oddělení. Na výzkumu v oblasti formální verifikace spolupracují také přední kapacity z prestižních universit po celém světě.

Nástroj MONA [1] je primárně určen pro rozhodování predikátů nad logikami WS1S a WS2S. Pro tento účel jsou jako nástroj používány *konečné stromové automaty* [2], přičemž knihovnu pro práci s nimi lze používat také zcela samostatně. Mezi oblasti, kde tato knihovna nachází využití, patří oblast formální verifikace, kde konečné stromové automaty nacházejí efektivní využití například pro reprezentaci dynamických datových struktur [3, 4]. Bez nadsázky můžeme říci, že nástroj MONA poskytuje knihovny, které jsou pro práci s konečnými stromovými automaty nejen v oblasti formální verifikace tím nejlepším, co je v současné době dostupné. Ovšem velikou nevýhodou těchto knihoven je přílišná složitost tvorby vstupních souborů, které obsahují předpis konečných stromových automatů. Jednou z variant konečných stromových automatů, která je v praxi využívána jako formální model XML schématu, je *hedge automata** [2]. Tato varianta se od konečných stromových automatů odlišuje především tím, že není explicitně dána arita vstupních symbolů.

*Tento pojem zde nebude překládán.

Cíl práce

Cílem této práce je návrh způsobu exportu a importu konečných stromových automatů a stromových převodníků z/do nástroje MONA, tak aby byl v souladu s formálními definicemi těchto matematických modelů. Tento návrh je potřeba implementovat a doplnit jej o realizaci interaktivního rozhraní, které by případným uživatelům nástroje MONA usnadnilo manipulaci s konečnými stromovými automaty a stromovými převodníky. Dále je potřeba připravit sadu testů, která umožní automatické otestování správnosti implementace. Výsledek této práce by měl najít využití v rámci výzkumné skupiny VeriFIT, která se zabývá metodami automatizované analýzy a verifikace.

Členění práce

Struktura práce je rozdělena do následujících kapitol. Kapitola 2 (Teoretické základy) je úvodem do teoretického pozadí, které je nutné znát pro pochopení formalismu konečných stromových automatů. V kapitole 3 (Dostupné knihovny pro práci se stromovými automaty) jsou představeny existující implementace knihoven pro práci s konečnými stromovými automaty. Většina této kapitoly je věnována nástroji MONA a jeho knihovnám s důrazem kladeným na osvětlení použitých principů a některých významných implementačních detailů. Kapitola 4 (Současný formát vstupu a výstupu knihoven MONA) obsahuje detailní popis formátu, ve kterém je nutné zadávat konečné stromové automaty do nástroje MONA. Návrh implementovaného nástroje je prezentován v kapitole 5 (Návrh implementace). Následuje kapitola 6 (Realizace a testování), jejímž obsahem je pojednání o samotné implementační části diplomové práce. Kapitola 7 (Zhodnocení) obsahuje zhodnocení současného stavu práce, vyznačení vlastního přínosu a úvahu nad dalším vývojem; uzavírá celou práci.

Kapitola 2

Teoretické základy

Tato kapitola si klade za cíl seznámení se základními definicemi z oblasti stromových automatů převzatými z [2]. Důkazy uváděných teorémů lze nalázt v [2]. Nejprve se podíváme na termy nad ohodnocenou abecedou a stromy, následovat bude popis stromových automatů a přehled uzávěrových vlastností stromových jazyků. Dále budou uvedeny některé rozhodnutelné problémy stromových jazyků. Závěr kapitoly je věnován stromovým převodníkům.

2.1 Termy a stromy

V této podkapitole budou uvedeny termy nad ohodnocenou abecedou a stromy.

2.1.1 Termy nad ohodnocenou abecedou

Abeceda Σ je konečná množina symbolů. Σ nazýváme *ohodnocená abeceda*, je-li definována *ohodnocovací funkce* $\# : \Sigma \longmapsto \mathbb{N}$ (kde $\mathbb{N}$ označuje množinu všech kladných celých čísel včetně nuly), která každému symbolu z Σ přiřazuje *aritu*. Pro každé $k \in \mathbb{N}$ je definována množina $\Sigma_k \subseteq \Sigma$, tj. množina všech symbolů s aritou rovnou k. Platí tedy vztah $\forall f \in \Sigma_k : \#(f) = k$. Symboly z množiny Σ_0 označujeme *konstanty* (nebo též nulární symboly). Předpokládáme, že Σ obsahuje alespoň jednu konstantu. V rámci příkladů budu využívat zkrácenou formu zápisu ohodnocení symbolů, kdy je arita symbolu naznačena pomocí kulatých závorek a čárek (např. $\mathtt{nil}$ je konstanta a $f_{(,)}$ je binární symbol f).

Mějme spočetnou množinu konstant $\mathcal{X}$, jejíž prvky nazýváme *proměnné*. Předpokládejme, že množiny Σ a $\mathcal{X}$ jsou navzájem disjunktní. $T(\Sigma, \mathcal{X})$ je množinou *termů* nad ohodnocenou abecedou Σ a množinou proměnných $\mathcal{X}$. Množinu termů $T(\Sigma, \mathcal{X})$ lze definovat takto:

- $\Sigma_0 \in T(\Sigma, \mathcal{X})$
- $\mathcal{X} \in T(\Sigma, \mathcal{X})$
- $\forall k > 0 : f \in \Sigma_k \wedge t_1, \ldots, t_k \in T(\Sigma, \mathcal{X}) \overset{\text{def}}{\Longleftrightarrow} f(t_1, \ldots, t_k) \in T(\Sigma, \mathcal{X})$

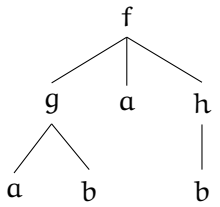
Pokud platí $\mathcal{X} = \emptyset$, pak se místo $T(\Sigma, \emptyset)$ používá zjednodušený zápis $T(\Sigma)$. Prvky množiny $T(\Sigma)$ označujeme jako *základní termy*. Term $t \in T(\Sigma, \mathcal{X})$ je *lineární* právě tehdy, když se v něm každá proměnná vyskytuje nejvýše jednou. Při vícenásobném výskytu jedné proměnné v termu $t \in T(\Sigma, \mathcal{X})$ říkáme, že t je *nelineární*.

2.1.2 Stromy

Na term $t \in T(\Sigma, \mathcal{X})$ můžeme přirozeně nahlížet jako na uspořádaný, ohodnocený strom, jehož listy jsou označeny buď proměnnými nebo konstantami, a nelistové uzly s k syny jsou označeny symboly s aritou rovnou k. Dále budeme zaměňovat pojmy term a strom.

Příklad 2.1 Vztah mezi termy a stromy

Nechť $\Sigma = \{f_{(,,)}, g_{(,)}, h_{()}, a, b\}$ je ohodnocená abeceda a $\mathcal{X} = \{x_1, x_2\}$ je množina proměnných. Term $g(x_2, x_1)$ je lineární, ale term $g(x_1, x_1)$ je nelineární. Základní term $f(g(a, b), a, h(b))$ lze graficky znázornit takto:



Alternativně lze na term $t \in T(\Sigma, \mathcal{X})$ nahlížet jako na parciální funkci $t : \mathbb{N}^* \longmapsto \Sigma \cup \mathcal{X}$, jejíž definiční obor označme $\mathsf{Pos}(t)$. Množina $\mathsf{Pos}(t)$ je neprázdná množina, pro kterou platí následující vztahy:

- $pd \in \mathsf{Pos}(t) \Rightarrow p \in \mathsf{Pos}(t)$, kde $p \in \mathbb{N}^*$, $d \in \mathbb{N}$ a $t \in T(\Sigma, \mathcal{X})$ (tato vlastnost je v anglické literatuře označována jako *prefix closure*),
- $\forall p \in \mathsf{Pos}(t) : (t(p) \in \Sigma_k \wedge k > 0) \Rightarrow \{d \mid pd \in \mathsf{Pos}(t)\} = \{1, \ldots, k\}$,
- $\forall p \in \mathsf{Pos}(t) : (t(p) \in \Sigma_0 \cup \mathcal{X}) \Rightarrow \{d \mid pd \in \mathsf{Pos}(t)\} = \emptyset$.

Prvky z množiny $\mathsf{Pos}(t)$ nazýváme *pozice*. Kořenový symbol termu t, označený $\mathsf{Head}(t)$, je definován na základě vztahu $\mathsf{Head}(t) = t(\varepsilon)$, kde ε chápeme jako prázdný řetězec přirozených čísel. Pod pojmem *listová pozice* chápeme pozici p takovou, že $\forall d \in \mathbb{N} : pd \notin \mathsf{Pos}(t)$. *Množinu listových pozic* termu t označíme $\mathsf{FPos}(t)$. Označme $\mathsf{NFPos}(t) = \{p \in \mathsf{Pos}(t) \mid \exists d \in \mathbb{N} : pd \in \mathsf{Pos}(t)\}$ jako *množinu nelistových pozic*. Každou pozici p v termu t, pro kterou platí $t(p) \in \mathcal{X}$, nazýváme *pozice proměnné*. Množinu pozic proměnných termu t označíme $\mathsf{VPos}(t)$.

Zápisem $t|_p$ značíme podterm, který se nachází na pozici p termu $t \in T(\Sigma, \mathcal{X})$. Podterm je definován takto:

- $\mathsf{Pos}(t|_p) = \{d \mid pd \in \mathsf{Pos}(t)\}$,
- $\forall d \in \mathsf{Pos}(t|_p) : t|_p(d) = t(pd)$.

Zápis $t[u]_p$ značí term, který získáme tím, že v původním termu t provedeme záměnu – místo podtermu $t|_p$ je u.

Uspořádání podtermů $\trianglerighteq$ je definováno takto: $t \trianglerighteq t' \overset{\text{def}}{\Longleftrightarrow} t'$ je podterm termu t. Setkat se můžeme také s ostrým uspořádáním podtermů $\triangleright$ s následující definicí: $t \triangleright t' \overset{\text{def}}{\Longleftrightarrow} t \trianglerighteq t' \wedge t \neq t'$.

Množina termů F je *uzavřená* právě tehdy, když je uzavřená vůči uspořádání podtermů, což lze vyjádřit vztahem $\forall t \in F : t \triangleright t' \Rightarrow t' \in F$.

Nad termy je definována dvojice funkcí. První funkce slouží pro zjištění *velikosti termu* t, značíme ji $\|t\|$ a definice je následující:

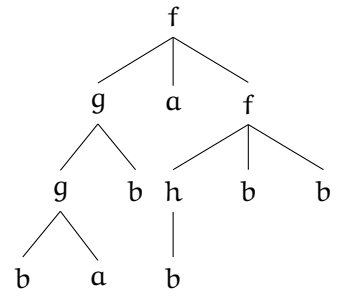
$$\|t\| = \begin{cases} 0 & t \in \mathcal{X} \\ 1 & t \in \Sigma_0 \\ 1 + \sum\limits_{i \in \{1,\ldots,k\}} \|t|_i\| & \mathsf{Head}(t) \in \Sigma_k \end{cases}$$

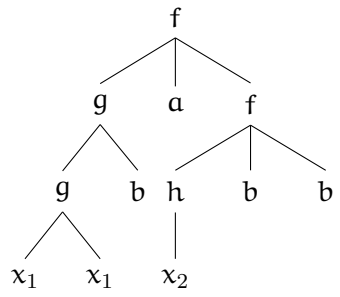
Druhou funkcí je *výška termu* t, $\mathsf{Height}(t)$, jejíž definice je:

$$\mathsf{Height}(t) = \begin{cases} 0 & t \in \mathcal{X} \\ 1 & t \in \Sigma_0 \\ 1 + \max\left\{\mathsf{Height}(t|_i) \mid i \in \{1,\ldots,k\}\right\} & \mathsf{Head}(t) \in \Sigma_k \end{cases}$$

Příklad 2.2 Přehled vlastností termů

Mějme abecedu $\Sigma = \{f_{(,,)}, g_{(,)}, h_{()}, a, b\}$ a množinu proměnných $\mathcal{X} = \{x_1, x_2\}$. Uvažujme termy $t = f(g(g(b,a),b),a,f(h(b),b,b))$ a $t' = f(g(g(x_1,x_1),b),a,f(h(x_2),b,b))$. Grafická reprezentace termů je následující:

$t =$ 

$t' =$ 

Následuje přehled vlastností termů t a t':

- Kořenové symboly: $\mathsf{Head}(t) = \mathsf{Head}(t') = f$.
- Množiny listových pozic: $\mathsf{FPos}(t) = \mathsf{FPos}(t') = \{111, 112, 12, 2, 311, 32, 33\}$.
- Množiny nelistových pozic: $\mathsf{NFPos}(t) = \mathsf{NFPos}(t') = \{\varepsilon, 1, 11, 3, 31\}$.
- Množiny pozic proměnných: $\mathsf{VPos}(t) = \emptyset$, $\mathsf{VPos}(t') = \{111, 112, 311\}$.
- Ukázky podtermů: $t|_2 = a$, $t|_{31} = h(b)$.
- Ukázka substituce podtermu: $t[a]_3 = f(g(g(b,a),b),a,a)$.
- Výšky termů: $\mathsf{Height}(t) = 4$, $\mathsf{Height}(t') = 3$.
- Velikosti termů: $\|t\| = 12 = |\mathsf{FPos}(t) \cup \mathsf{NFPos}(t)|$, $\|t'\| = 9$.

2.1.3 Substituce

Substituce je zobrazení $\sigma : \mathcal{X} \longmapsto \mathsf{T}(\Sigma, \mathcal{X})$ takové, že pouze konečné množství proměnných není zobrazeno na sebe sama. Definičním oborem substituce σ je podmnožina

proměnných $x \in \mathcal{X}$ takových, že $x \neq \sigma(x)$. Pojmem *základní substituce* rozumíme zobrazení $\sigma : \mathcal{X} \longmapsto T(\Sigma)$. Substituce $\{x_1 \leftarrow t_1, \ldots, x_k \leftarrow t_k\}$ zobrazuje $x_i \in \mathcal{X}$ na $t_i \in T(\Sigma, \mathcal{X})$ pro $\forall i \in \{1, \ldots, k\}$. Definiční obor substituce můžeme rozšířit na $T(\Sigma, \mathcal{X})$, přičemž se řídíme vztahem:

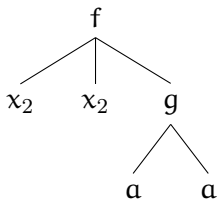
$$\forall f \in \Sigma_k, \forall t_1, \ldots, t_k \in T(\Sigma, \mathcal{X}) : \sigma(f(t_1, \ldots, t_k)) = f(\sigma(t_1), \ldots, \sigma(t_k)).$$

V dalším textu budeme pod pojmem substituce chápat substituci s definičním oborem rozšířeným na $T(\Sigma, \mathcal{X})$. Substituce budeme v některých případech zapisovat pomocí postfixové notace, a tak chápeme zápis $t\sigma$ jako výsledek aplikace substituce σ na term t.

Příklad 2.3 Substituce

Mějme abecedu $\Sigma = \{f_{(,,)}, g_{(,)}, a\}$ a množinu proměnných $\mathcal{X} = \{x_1, x_2\}$. Uvažujme term $t = f(x_1, x_1, x_2)$, a substituci $\sigma = \{x_1 \leftarrow x_2, x_2 \leftarrow g(a, a)\}$. Pak platí:

$$t\sigma = t\{x_1 \leftarrow x_2, x_2 \leftarrow g(a, a)\} = f(x_2, x_2, g(a, a)) =$$



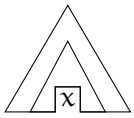
2.1.4 Kontexty

Nechť je $\mathcal{X}_k$ množinou k proměnných. Lineární term $C \in T(\Sigma, \mathcal{X}_k)$ nazýváme *kontext* a výraz $C[t_1, \ldots, t_k]$, kde $t_1, \ldots, t_k \in T(\Sigma)$, označuje term z množiny $T(\Sigma)$, který jsme získali tím, že jsme v C zaměnili t_i za proměnnou x_i pro každé $i \in \{1, \ldots, k\}$. Pak platí $C[t_1, \ldots, t_k] = C\{x_1 \leftarrow t_1, \ldots, x_k \leftarrow t_k\}$. Množinu všech kontextů nad $(x_1, \ldots, x_k)$ označujeme $\mathcal{C}^k(\Sigma)$. Pro $k = 1$ se místo $\mathcal{C}^k(\Sigma)$ používá označení $\mathcal{C}(\Sigma)$.

Množinu kontextů obsahujících jedinou proměnnou značíme $\mathcal{C}(\Sigma)$. Kontext je *triviální* právě tehdy, když je redukován na proměnnou. Pro daný kontext $C \in \mathcal{C}(\Sigma)$ platí, že následující jsou kontexty z $\mathcal{C}(\Sigma)$:

- C^0 je triviální kontext,
- C^1 je ekvivalentním zápisem k C,
- $C^n = C^{n-1}[C]$ pro $n > 1$.

Obrázek 2.1 Ilustrace mocnin kontextu $C \in \mathcal{C}(\Sigma)$ nad proměnnou x

$C^0 = x$ $\quad$ $C^1 = C =$  $\quad$ $C^2 = C^1[C] =$ 

Mocniny kontextu nad jedinou proměnnou jsou naznačeny na obr. 2.1. S mocninami kontextu se dále setkáme v definici Pumping Lemmatu pro regulární stromové jazyky (viz podkapitola 2.3.1).

2.2 Modely regulárních stromových jazyků

V rámci této kapitoly bude představeno několik různých variant konečných stromových automatů, jenž si uvedeme jako formální modely třídy jazyků označované regulární stromové jazyky.

2.2.1 Stromové jazyky a konečné stromové automaty

Nejprve si nadefinujeme *nedeterministický konečný stromový automat* (NFTA) nad Σ, který pracuje zdola nahoru. NFTA je čtveřice $M = (Q, \Sigma, Q_f, \Delta)$, kde:

Q je konečná množina stavů,

Σ je ohodnocená abeceda (označovaná též jako vstupní abeceda),

$Q_f \subseteq Q$ je množina koncových stavů,

Δ je množina přechodových pravidel tvaru:

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n))$$

kde $n \geqslant 0$, $f \in \Sigma_n$, $q, q_1, \ldots, q_n \in Q$ a $x_1, \ldots, x_n \in \mathcal{X}$. Kromě tohoto zápisu se můžeme setkat také se zjednodušeným zápisem přechodových pravidel, při jehož použití jsou uváděny pouze symboly a stavy. Ukázka zjednodušeného tvaru pravidla:

$$f(q_1, \ldots, q_n) \to q$$

kde kde $n \geqslant 0$, $f \in \Sigma_n$ a $q, q_1, \ldots, q_n \in Q$. V rámci této práce zůstaneme u původního tvaru – proměnné v zápisu pravidel zachováme.

Činnost konečného stromového automatu M je modelována pomocí binární relace na množině termů $\vdash_M : T(\Sigma \cup Q) \times T(\Sigma \cup Q)$. *Relace přechodu* $\vdash_M$ je definována:

$$t \vdash_M t' \quad \overset{\text{def}}{\iff} \quad \begin{cases} \exists C \in \mathcal{C}(\Sigma \cup Q) & \wedge \\ \exists t_1, \ldots, t_n \in T(\Sigma) & \wedge \\ \exists f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)) \in \Delta & \wedge \\ t = C[f(q_1(t_1), \ldots, q_n(t_n))] & \wedge \\ t' = C[q(f(t_1, \ldots, t_n))] \end{cases}$$

Relace $\vdash_M^*$ je reflexivně tranzitivní uzávěr relace $\vdash_M$. Zápis $t \vdash_M^* t'$ znamená, že NFTA transformuje term t po konečném počtu přechodů na term t'.

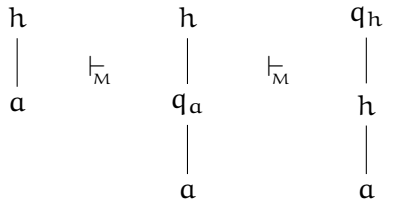
NFTA nemá definován žádný počáteční stav. Zpracování základního termu pomocí konečného stromového automatu začíná u listů daného termu a pokračuje směrem vzhůru, přičemž induktivně podtermům *přiřazuje stavy* (v angličtině se používá termín *labeling*). Počáteční stav je u NFTA zastoupen přechodovými pravidly tvaru $a \to q(a)$, kde $a \in \Sigma_0$ a $q \in Q$. Pokud jsou všem přímým podtermům $t_1, \ldots, t_n$ termu $t = f(t_1, \ldots, t_n)$ přiřazeny stavy $q_1, \ldots, q_n$, pak bude termu t přiřazen stav q na základě přechodového pravidla $f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)) \in \Delta$, kde jsou za proměnné $x_1, \ldots, x_n$ substituovány podtermy $t_1, \ldots, t_n$.

Základní term $t \in T(\Sigma)$ je *přijímán* konečným stromovým automatem $M = (Q, \Sigma, Q_f, \Delta)$ právě tehdy, když:

$$t \vdash_M^* q_f(t)$$

Příklad 2.4 Činnost konečného stromového automatu

Uvažujme konečný stromový automat $M = (Q, \Sigma, Q_f, \Delta)$, kde $\Sigma = \{h^{(1)}, a\}$, $Q = \{q_a, q_h\}$, $Q_f = \{q_h\}$ a $\Delta = \{a \to q_a(a), h(q_a(x)) \to q_h(h(x))\}$. Na termu $t = h(a)$ si ukážeme aplikaci relace $\vdash_M$:



, což lze zkrátit zápisem $t \vdash_M^* q_h(t)$.

pro $q_f \in Q_f$. Stromový jazyk přijímaný konečným stromovým automatem M je označován $L(M)$ a je definován jako množina všech základních termů přijímaných konečným stromovým automatem M:

$$L(M) = \{t \mid t \vdash_M^* q_f(t) \wedge q_f \in Q_f\}$$

Množinu základních termů L nazýváme *regulární stromový jazyk*, pokud existuje konečný stromový automat M takový, že $L = L(M)$. Pokud dva (nebo více) konečných stromových automatů přijímá stejný stromový jazyk, jsou *ekvivalentní*.

Alternativně lze množinu přechodových pravidel Δ definovat jako přepisovací systém základních termů. Tvar přechodových pravidel pak bude následující:

$$f(q_1, \ldots, q_n) \to q$$

kde $n \geqslant 0$, $f \in \Sigma_n$ a $q, q_1, \ldots, q_n \in Q$. Použití tohoto tvaru přepisovacích pravidel přináší dvě zásadní změny. Za prvé, konečný stromový automat již při své činnosti nezachovává strukturu termu. Postupně ztrácí podtermy, kterým již přiřadil stav. Za druhé, NFTA přijme term t, když:

$$t \vdash_M^* q_f$$

kde $q_f \in Q_f$. Mějme NFTA M a základní term t. Zavedeme funkci $r : Pos(t) \longmapsto Q$ takovou, že pro všechny pozice p platí:

$$\forall i \in \{1, \ldots, n\} : t(p) = f \in \Sigma_n \wedge r(p) = q \wedge r(pi) = q_i \overset{\text{def}}{\iff} f(q_1, \ldots, q_n) \to q \in \Delta$$

Funkce r se nazývá *běh* konečného stromového automatu M na termu t a slouží pro zachycení historie zpracování termu t pomocí konečného stromového automatu M. Každému symbolu v termu přiřazuje stav, kterým byl označen. Běh r NFTA M na termu t je *úspěšný* právě tehdy, když $r(\varepsilon) = q_f$, kde $q_f \in Q_f$. Běh konkrétního konečného stromového automatu je uveden v příkladu 2.5.

Konečný stromový automat je *úplný*, pokud existuje alespoň jedno pravidlo tvaru:

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)) \in \Delta$$

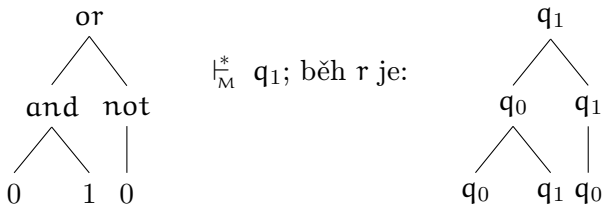
pro každé $n \geqslant 0$, $f \in \Sigma_n$ a $q_1, \ldots, q_n \in Q$. Stav $q \in Q$ je *dostupný* právě tehdy, když existuje základní term t takový, že $t \vdash_M^* q(t)$. NFTA je *redukovaný*, pokud jsou všechny jeho stavy dostupné.

Příklad 2.5 Konečný stromový automatu pro vyhodnocení výrazů nad Booleovou algebrou

Mějme abecedu $\Sigma = \{\mathtt{or}_{(,)}, \mathtt{and}_{(,)}, \mathtt{not}_{()}, 1, 0\}$, kde konstanta 1 reprezentuje pravdivou hodnotu a konstanta 0 značí nepravdivou hodnotu. Uvažujme konečný stromový automat $M = (Q, \Sigma, Q_f, \Delta)$, kde $Q = \{q_1, q_0\}$, $Q_f = \{q_1\}$ a $\Delta =$

$$\begin{array}{lllll}
\{ & 1 \to q_1, & & 0 \to q_0, & \\
& \mathtt{not}(q_1) \to q_0 & & \mathtt{not}(q_0) \to q_1 & \\
& \mathtt{or}(q_1, q_1) \to q_1 & & \mathtt{or}(q_1, q_0) \to q_1 & \\
& \mathtt{or}(q_0, q_1) \to q_1 & & \mathtt{or}(q_0, q_0) \to q_0 & \\
& \mathtt{and}(q_1, q_1) \to q_1 & & \mathtt{and}(q_1, q_0) \to q_0 & \\
& \mathtt{and}(q_0, q_1) \to q_0 & & \mathtt{and}(q_0, q_0) \to q_0 & \}.
\end{array}$$

Běh NFTA M nad základním termu $t = \mathtt{or}(\mathtt{and}(0,1), \mathtt{not}(0))$:



Na základní term nad ohodnocenou abecedou Σ lze nahlížet jako na výraz Booleovy algebry a běh NFTA M nad daným základním termem lze chápat jako vyhodnocení odpovídajícího výrazu. Stromový jazyk přijímaný konečným stromovým automatem M je množinou všech pravdivých výrazů Booleovy algebry nad Σ. Běh r je úspěšný, z čehož vyplývá, že výraz reprezentovaný základním termem t je pravdivý.

2.2.2 Nedeterministické konečné stromové automaty s ε-pravidly

Definici *nedeterministického konečného stromového automatu s ε-pravidly* se od definice NFTA odlišuje pouze tím, že množina přechodových pravidel může navíc obsahovat pravidla tvaru $q \to q'$, jenž označujeme *ε-pravidla*. Při aplikaci ε-pravidla dochází ke změně stavu, aniž by byl zpracován vstupní symbol.

Teorém 2.1. (Ekvivalence NFTA s ε-pravidly a bez nich). *Pokud je stromový jazyk $\mathcal{L}$ přijímán NFTA s ε-pravidly, pak existuje NFTA bez ε-pravidel přijímající jazyk $\mathcal{L}$.*

Vzhledem k tomu, že platí teorém 2.1, není potřeba rozlišovat mezi NFTA a NFTA s ε-pravidly. Použití NFTA s ε-pravidly je výhodné, protože díky ε-pravidlům lze dosáhnout zjednodušení některých konstrukcí a důkazů.

2.2.3 Deterministické konečné stromové automaty

Deterministický konečný stromový automat (DFTA) je NFTA bez takových přechodových pravidel, která mají stejnou levou stranu, a bez ε-pravidel v Δ. DFTA je, narozdíl od NFTA, *jednoznačný*, což znamená, že pro každý základní term t existuje nejvýše jeden běh a zároveň existuje nejvýše jeden stav $q \in Q$, pro který platí $t \vdash^*_M q$.

Teorém 2.2. (Ekvivalence NFTA a DFTA). *Mějme regulární stromový jazyk $\mathcal{L}$. Pak existuje DFTA, který přijímá $\mathcal{L}$.*

V definici deterministických konečných stromových automatů je možné použít *funkci přechodu* δ namísto množiny přechodových pravidel Δ. V takovém případě je DFTA $M = (Q, \Sigma, Q_f, \delta)$, kde:

$$\delta : \bigcup_n \Sigma_n \times Q^n \longmapsto Q$$

Pomocí funkce přechodu lze definovat *funkci přiřazení stavu* (v angličtině se používá termín *labeling function*), což je zobrazení $\hat{\delta} : T(\Sigma) \longmapsto Q$ definované pomocí indukce následujícím vztahem:

$$\hat{\delta}(f(t_1, \ldots, t_n) = \delta(f, \hat{\delta}(t_1), \ldots, \hat{\delta}(t_n)))$$

kde $f \in \Sigma_n$ a $t_1, \ldots, t_n \in T(\Sigma)$. Funkce δ a $\hat{\delta}$ mají z praktického hlediska stejný význam, a tak mohou být libovolně zaměňovány.

Z teorému 2.2 vyplývá, že DFTA a NFTA přijímají stejnou množinu stromových jazyků. Není-li nutné mezi nimi rozlišovat, používá se souhrnné označení *konečný stromový automat* (zkráceně FTA).

2.3 Vlastnosti regulárních stromových jazyků

V rámci této podkapitoly je věnován větší prostor třídě regulárních stromových jazyků.

2.3.1 Pumping lemma pro regulární stromové jazyky

Podobně jako pro regulární a bezkontextové jazyky, existuje také Pumping lemma pro regulární stromové jazyky. Toto lemma je velmi užitečné pro dokazování, že určité množiny základních termů nespadají do třídy regulárních stromových jazyků, tj. neexistuje takový konečný stromový automat, který by takové množiny základních termů přijímal. Pumping lemmu lze dále použít pro řešení některých rozhodnutelných problémů, mezi které patří např. problém prázdnosti a problém konečnosti regulárních stromových jazyků.

Pumping Lemma pro regulární stromové jazyky. *Nechť $\mathcal{L}$ je regulární stromový jazyk. Pak existuje konstanta $k > 0$, pro kterou platí: pro každý základní term $t \in \mathcal{L}$ takový, že $\mathsf{Height}(t) > k$, existuje kontext $C \in \mathcal{C}(\Sigma)$, netriviální kontext $C' \in \mathcal{C}(\Sigma)$ a základní term u takový, že $t = C[C'[u]]$ a zároveň $\forall n \geqslant 0 : C[C'^n[u]] \in \mathcal{L}$.*

Příklad 2.6 Aplikace Pumping lemmatu (převzato z [2])

Mějme ohodnocenou abecedu $\Sigma = \{f_{(,)}, a\}$. Uvažujme stromový jazyk $L = \{t \mid t \in T(\Sigma) \wedge |\mathsf{Pos}(t)| \text{ je prvočíslem}\}$. Můžeme dokázat, že L nepatří do třídy regulárních stromových jazyků. Pro všechna $k > 0$ uvažujme term $t \in L$ takový, že $\mathsf{Height}(t) > k$. Pro všechny kontexty C, netriviální kontexty C' a termy u takové, že $t = C[C'[u]]$, existuje takové n, pro které platí $C[C'^n[u]] \notin L$.
Pozn.: Vzhledem k povaze Pumping lemmatu a relativně nízkému počtu prvočísel je výše uvedené tvrzení zřejmé.

2.3.2 Myhill-Nerodův Teorém pro stromové jazyky

Myhill-Nerodův Teorém patří mezi jedny z nejznámějších teorémů z teorie konečných automatů. Tento teorém ukazuje několik různých pohledů na regulární jazyky a existuje pro

něj celá řada užitečných aplikací. Jedním z důsledků, které z tohoto teorému vyplývají, je ten, že pro každý regulární jazyk nad konečnou abecedou existuje unikátní deterministický konečný automat s nejmenším možným počtem stavů. Myhill-Nerodův Teorém lze přímo zobecnit na konečné automaty pracující nad konečnými stromy.

Relace ekvivalence $\equiv$ na $T(\Sigma)$ je *kongruencí* na $T(\Sigma)$, pokud pro každý symbol $f \in \Sigma_n$ platí:

$$\left(\forall i \in \{1, \ldots, n\} : u_i \equiv v_i\right) \Rightarrow f(u_1, \ldots, u_n) \equiv f(v_1, \ldots, v_n)$$

Kongruence *konečného indexu* je taková kongruence, pro kterou platí, že rozklad podle této kongruence má konečné množství tříd. Ekvivalentní k této definici je následující definice: kongruence je relace ekvivalence uzavřená pod kontextem, tzn. pro každý kontext $C \in \mathcal{C}(\Sigma)$ platí $u \equiv v \Rightarrow C[u] \equiv C[v]$. Uvažujme regulární stromový jazyk $\mathcal{L}$, pak binární relace $\equiv_{\mathcal{L}}$ na $T(\Sigma)$ je definována takto: $u \equiv_{\mathcal{L}} v$ platí, když pro všechny kontexty $C \in \mathcal{C}(\Sigma)$ je splněn následující vztah:

$$C[u] \in \mathcal{L} \iff C[v] \in \mathcal{L}$$

Myhill-Nerodův Teorém pro strromové jazyky. *Následující tři tvrzení jsou navzájem ekvivalentní:*

(i) $\mathcal{L}$ *je regulární stromový jazyk,*

(ii) $\mathcal{L}$ *je sjednocením některých ekvivalenčních tříd rozkladu definovaného kongruencí konečného indexu,*

(iii) relace $\equiv_{\mathcal{L}}$ *je kongruence konečného indexu.*

Na základě MyHill-Nerodova Teorému lze dokázat, že pro každý regulární stromový jazyk lze vytvořit deterministický konečný stromový automat, který je unikátní až na pojmenování stavů. Jedná se o důkaz implikace *(iii)* $\Rightarrow$ *(i)*, jehož postup je uveden v kapitole 1.5 [2].

2.3.3 Uzávěrové vlastnosti regulárních stromových jazyků

Teorém 2.3. *Třída regulárních stromových jazyků je uzavřená vůči sjednocení.*

Mějme následující dvojici konečných stromových automatů: $M_1 = (Q_1, \Sigma, Q_{f1}, \Delta_1)$ a $M_2 = (Q_2, \Sigma, Q_{f2}, \Delta_2)$, přičemž bez ztráty na obecnosti můžeme předpokládat, že $Q_1 \cap Q_2 = \emptyset$. Pokud by tato podmínka splněna nebyla, stačí přejmenovat stavy jednoho z konečných stromových automatů. Nyní můžeme sestrojit FTA $M = (Q, \Sigma, Q_f, \Delta)$, který je definován takto: $Q = Q_1 \cup Q_2$, $Q_f = Q_{f1} \cup Q_{f2}$ a $\Delta = \Delta_1 \cup \Delta_2$. Platí $L(M) = L(M_1) \cup L(M_2)$, avšak M je nedeterministický, i když jsou M_1 a M_2 deterministické.

Předpokládejme, že M_1 a M_2 jsou úplné*. Sestrojíme konečná stromový automat $M = (Q, \Sigma, Q_f, \Delta)$ přijímající jazyk $L(M) = L(M_1) \cup L(M_2)$, kde $Q = Q_1 \times Q_2$, $Q_f = (Q_{f1} \times Q_2) \cup (Q_1 \times Q_{f2})$ a $\Delta = \Delta_1 \times \Delta_2$, kde:

$$\begin{aligned}\Delta_1 \times \Delta_2 = \big\{ & f((q_1, q_1'), \ldots, (q_n, q_n')) \to (q, q') \mid \\ & f(q_1, \ldots, q_n) \to q \in \Delta_1 \wedge f(q_1', \ldots, q_n') \to q' \in \Delta_2 \big\}\end{aligned}$$

Tento postup pro sjednocení dvou konečných stromových automatů zachovává determinismus, tj. pokud jsou M_1 a M_2 deterministické, pak také M je deterministický.

*Každý NFTA lze převést na kompletní NFTA tím, že se doplní chybějící přechody tak, aby směřovaly do nekoncového stavu, tzv. *sink* stavu.

Teorém 2.4. *Třída regulárních stromových jazyků je uzavřená vůči doplňku.*

Mějme úplný DFTA $M = (Q, \Sigma, Q_f, \Delta)$, pak $L(M)$ je regulární stromový jazyk. Tím, že provedeme doplněk množiny koncových stavů, získáme deterministický konečný stromový automat přijímající doplněk jazyka $L(M)$ nad $T(\Sigma)$. Formálně zapsáno: DFTA $\overline{M} = (Q, \Sigma, Q \setminus Q_f, \Delta)$ přijímá jazyk $\overline{L(M)}$, kde $\overline{L}$ označuje doplněk množiny L nad $T(\Sigma)$.

Teorém 2.5. *Třída regulárních stromových jazyků je uzavřená vůči průniku.*

Uzavřenost regulárních stromových jazyků vůči průniku přímo vychází z uzavřenosti vůči sjednocení a doplňku. Pro konstrukci můžeme použít přímo De Morganova zákona:

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

avšak je vhodnější použít přímý algoritmus. Mějme konečné stromové automaty $M_1 = (Q_1, \Sigma, Q_{f1}, \Delta_1)$ a $M_2 = (Q_2, \Sigma, Q_{f2}, \Delta_2)$. Uvažujme FTA $M = (Q, \Sigma, Q_f, \Delta)$ definovaný takto: $Q = Q_1 \times Q_2$, $Q_f = Q_{f1} \times Q_{f2}$ a $\Delta = \Delta_1 \times \Delta_2$. Pro M platí vztah $L(M) = L(M_1) \cap L(M_2)$. Tento postup zachovává determinismus.

2.3.4 Stromový homomorfismus

Třída regulárních jazyků je uzavřená vůči homomorfismu a inverznímu homomorfismu, avšak u stromových regulárních jazyků je situace trochu komplikovanější. Stromové regulární jazyky jsou uzavřeny pouze vůči určité podtřídě homomorfismu, tzv. lineárnímu homomorfismu. Lineární homomorfismus zachovává strukturu termu, tzn. nedovoluje duplikaci vkládaných podtermů. Následuje definice stromového homomorfismu, jak budeme označovat homomorfismus pro regulární stromové jazyky.

Mějme dvě ohodnocené abecedy Σ a Σ', které nutně nemusejí být disjunktní. Pro každé $n > 0$ takové, že Σ obsahuje symbol arity n, definujme množinu proměnných $\mathcal{X}_n = \{x_1, \ldots, x_n\}$, pro kterou platí $\Sigma \cup \mathcal{X}_n = \emptyset$ a $\Sigma' \cup \mathcal{X}_n = \emptyset$. Nechť h_Σ je zobrazení takové, že k symbolu $f \in \Sigma$ s aritou n přiřazuje term $t_f \in T(\Sigma', \mathcal{X}_n)$. *Stromový homomorfismus* $h : T(\Sigma) \longmapsto T(\Sigma')$, který je určený zobrazením h_Σ, je definován následovně:

- $\forall a \in \Sigma_0 : h(a) = t_a \in T(\Sigma')$,
- $h(f(t_1, \ldots, t_n)) = t_f\{x_1 \leftarrow h(t_1), \ldots, x_n \leftarrow h(t_n)\}$.

kde zápisem $t_f\{x_1 \leftarrow h(t_1), \ldots, x_n \leftarrow h(t_n)\}$ označujeme výsledek aplikace substituce $\{x_1 \leftarrow h(t_1), \ldots, x_n \leftarrow h(t_n)\}$ na term t_f.

Stromový homomorfismus je *lineární* právě tehdy, když $\forall f \in \Sigma_n : h_\Sigma(f) = t_f$, kde $t_f \in T(\Sigma', \mathcal{X}_n)$ je lineární term. Homomorfismus h uvedený v příkladu 2.7 je lineární.

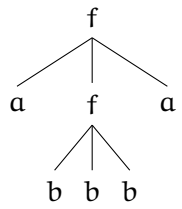
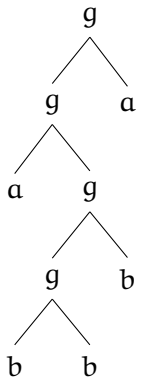
Teorém 2.6. (Lineární stromový homomorfismus zachovává strukturu regulárního stromového jazyku). *Nechť h je lineární stromový homomorfismus a $\mathcal{L}$ je regulární stromový jazyk, pak $h(\mathcal{L})$ je také regulární stromový jazyk.*

Pouze lineární stromový homomorfismus zachovává strukturu regulárního stromového jazyka, avšak pro lineární i nelineární homomorfismy platí teorém 2.7.

Teorém 2.7. (Inverzní stromový homomorfismus zachovává strukturu regulárního stromového jazyku). *Nechť h je stromový homomorfismus a $\mathcal{L}$ je regulární stromový jazyk, pak $h^{-1}(\mathcal{L})$ je také regulární stromový jazyk.*

Příklad 2.7 Lineární homomorfismus

Mějme ohodnocené abecedy $\Sigma = \{f_{(,,)}, a, b\}$ a $\Sigma = \{g_{(,)}, a, b\}$. Dále uvažujme stromový homomorfismus h určený zobrazením h_Σ, jehož předpis je: $h_\Sigma(f) = g(g(x_1, x_2), x_3)$, $h_\Sigma(a) = a$ a $h_\Sigma(b) = b$. Homomorfismus h budeme aplikovat na term $t = f(a, f(b, b, b), a)$:

Pokud $t =$  , pak $h(t) =$ 

Homomorfismus h definuje transformaci z ternárního stromu do binárního stromu.

2.3.5 Rozhodnutelné problémy regulárních stromových jazyků

Tato podkapitola obsahuje přehled některých rozhodnutelných problémů regulárních stromových jazyků a jejich složitosti. Pro odvození složitosti rozhodnutelných problémů je používán rámec RAM strojů.

Problém členství
Rozhodnutí, zdali je určitý základní term přijat daným konečným stromovým automatem. V tomto případě již je na vstupu rozhodovací procedury základní term i FTA. Tento problém může být rozhodnut v *lineárním čase* pro DFTA a v *polynomiálním čase* pro NFTA.

Problém neprázdnosti
Rozhodnutí, je-li jazyk přijímaný zadaným konečným automatem prázdný.
Problém (ne)prázdnosti lze rozhodnout v *lineárním čase*.

Problém neprázdnosti průniku
Rozhodnutí existence alespoň jednoho základního termu pro každý konečný stromový automat ze zadané množiny FTA.
Jedná se o *EXPTIME-úplný* problém.

Problém konečnosti
Rozhodnutí, jestli je jazyk přijímaný daným konečným stromovým automatem konečný, tj. jedná se o konečnou množinu základních termů.
Konečnost lze rozhodnout v *polynomiálním čase*.

Problém prázdnosti doplňku
Rozhodnutí, zdali daný konečný stromový automat přijímá množinu všech termů.
Pro DFTA je problém prázdnosti doplňku rozhodnutelný v *polynomiálním* čase, avšak pro NFTA je daný problém *EXPTIME-úplný*.

Problém ekvivalence
Rozhodnutí, jestli dvojice zadaných konečných stromových automatů přijímá stejný jazyk.
Problém ekvivalence je *rozhodnutelný* (a existuje více přístupů).

Problém singletonu
Rozhodnutí, jestli daný automat přijímá pouze jeden jediný základní term, tj. jazyk přijímaný daným konečným stromovým automatem je jednoprvkovou množinou (tzv. singletonem).
Tento problém je rozhodnutelný v *polynomiálním* čase.

V rámci této podkapitoly jsme si mohli všimnout toho, že složitost řešení rozhodnutelných problémů pro třídu regulárních stromových jazyků je vyšší při porovnání s třídou regulárních jazyků (tedy třídou $\mathcal{L}_3$ známou z Chomského hierarchie jazyků).

2.4 Konečné stromové automaty pracující shora dolů

Podkapitola 2.2.1 se zabývala konečnými stromovými automaty, které pracovali zdola nahoru (v angličtině se používá termín *bottom-up FTA*). Zde se budeme věnovat konečným stromových automatům, které pracují shora dolů (angl. *top-down FTA*).

Nedeterministický konečný stromový automat *pracující shora dolů* je čtveřice $M = (Q, \Sigma, I, \Delta)$, kde:

Q je konečná množina stavů,

Σ je ohodnocená abeceda,

$I \subseteq Q$ je množina počátečních stavů,

Δ je množina přechodových pravidel tvaru:

$$q(f(x_1, \ldots, x_n)) \to f(q_1(x_1), \ldots, q_n(x_n))$$

kde $n \geqslant 0$, $f \in \Sigma_n$, $q, q_1, \ldots, q_n \in Q$ a $x_1, \ldots, x_n \in \mathcal{X}$.

Pokud platí $n = 0$, tj. jedná se o konstantu a, přechodové pravidlo NFTA pracujícího shora dolů nabývá tvaru $q(a) \to a$. Automat pracující zdola nahoru začíná u kořene stromu a postupuje směrem dolů, přičemž induktivně přiřazuje stavy jednotlivým podtermům. Relaci přechodu NFTA pracujícího shora dolů lze jednoduše odvodit z příslušné definice NFTA pracujícího zdola nahoru. Stromový jazyk $L(M)$ přijímaný konečným stromovým automatem $M = (Q, \Sigma, I, \Delta)$ je množina všech základních termů t, pro které existuje počáteční stav $q_0 \in I$ takový, aby byl splněn vztah:

$$q_0(t) \overset{*}{\underset{M}{\vdash}} t$$

Teorém 2.8. Ekvivalence nedeterministických konečných stromových automatů pracujících zdola nahoru a shora dolů. *Třída jazyků přijímaných NFTA pracujícími shora dolů je třída regulárních stromových jazyků.*

Z teorému 2.8 vyplývá, že nedeterministické konečné stromové automaty mají stejnou vyjadřovací sílu bez ohledu na to, zda pracují zdola nahoru nebo shora dolů. Avšak, DFTA pracující shora dolů má striktně nižší vyjadřovací sílu nežli NFTA pracující shora dolů.

2.5 Stromové převodníky

Dosud jsme se zabývali konečnými stromovými automaty, což jsou modely regulárních stromových jazyků, tzn. reprezentují množinu základních termů nad danou ohodnocenou abecedou. V této podkapitole se budeme věnovat stromovým převodníkům, jež modelují určitou relaci nad termy. Stromové převodníky jsou ve své podstatě konečné stromové automaty realizující transformaci termů. Oproti konečným stromovým automatům jsou rozšířené o tzv. *výstupní abecedu* – stromový převodník provádí transformaci základních termů nad vstupní abecedou na základní termy nad výstupní abecedou.

Jedním z typických příkladů využití stromových převodníků je oblast překladačů programovacích jazyků. Výstupem parseru je derivační strom, který můžeme v daném kontextu považovat za term. Tento term je při překladu dále transformován, přičemž tuto transformaci lze realizovat právě pomocí stromového převodníku, který musí splňovat určité vlastnosti – musí zachovávat strukturu termu. Stromovým převodníkům zachovávajícím strukturu je věnován prostor na konci této podkapitoly, avšak nejprve začneme tím, že si stromové převodníky rozdělíme podle principu jejich činnosti na stromové převodníky pracující zdola nahoru a stromové převodníky pracující shora dolů.

2.5.1 Stromové převodníky pracující zdola nahoru

Nedeterministický stromový převodník pracující zdola nahoru je formálně definován jako pětice $\tau = (Q, \Sigma, \Sigma', Q_f, \Delta)$, kde:

Q je konečná množina stavů,

Σ je konečná neprázdná množina vstupních symbolů (vstupní abeceda) taková, že $\Sigma \cap Q = \emptyset$,

Σ' je konečná neprázdná množina výstupních symbolů (výstupní abeceda), pro kterou platí $\Sigma' \cap Q = \emptyset$,

$Q_f \subseteq Q$ je množina koncových stavů,

Δ je konečná množina převodních pravidel, která mohou být následujících dvou typů:

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(u)$$

kde $f \in \Sigma_n$, $q, q_1, \ldots, q_n \in Q$, $x_1, \ldots, x_n \in \mathcal{X}_n$ a $u \in T(\Sigma', \mathcal{X}_n)$, a ε-pravidlo:

$$q(x) \to q'(u)$$

kde $q, q' \in Q$, $x \in \mathcal{X}$ a $u \in T(\Sigma', \mathcal{X}_1)$.

Mějme dvojici termů $t, t' \in T(\Sigma \cup \Sigma' \cup Q)$. *Relace přechodu* $\vdash_\tau$ je definována takto:

$$t \vdash_\tau t' \quad \overset{\text{def}}{\iff} \quad \begin{cases} \exists f(q_1(x_1), \ldots, q_n(x_n)) \to q(u) \in \Delta & \wedge \\ \exists C \in \mathcal{C}(\Sigma \cup \Sigma' \cup Q) & \wedge \\ \exists u_1, \ldots, u_n \in T(\Sigma') & \wedge \\ t = C[f(q_1(u_1), \ldots, q_n(u_n))] & \wedge \\ t' = C[q(u\{x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n\})] \end{cases}$$

Relace $\vdash_\tau^*$ je reflexivně tranzitivní uzávěr relace $\vdash_\tau$. Relace R_τ indukovaná převodníkem τ je:

$$\tau = \{(t, t') \mid t \vdash_\tau^* q(t') \wedge t \in T(\Sigma) \wedge t' \in T(\Sigma') \wedge q \in Q_f\}$$

Převodníku, jehož Δ neobsahuje ani jedno ε-pravidlo, patří přídomek *bez ε-pravidel* (v angličtině se používá termín *ε-free*). Pokud jsou všechna převodní pravidla lineární (tj. na pravé straně převodního pravidla se smí každá proměnná vyskytovat nejvýše jednou), hovoříme o *lineárním* převodníku. Pro *neodmazávající* (volný překlad z anglického termínu *non-erasing*) převodník platí, že každé pravidlo má na pravé straně alespoň jeden symbol z Σ'. Pokud pro všechna pravidla $f(q_1(x_1), \ldots, q_n(x_n)) \to q(u)$ platí, že pro každé x_i, kde $1 \leqslant i \leqslant n$, se x_i vyskytuje alespoň jednou v u, pak je převodník *úplný*. Převodník je *deterministický* právě tehdy, když je bez ε-pravidel a neobsahuje dvě pravidla se stejnou levou stranou.

2.5.2 Stromové převodníky pracující shora dolů

Nedeterministický stromový převodník pracující shora dolů je formálně definován jako pětice $\tau = (Q, \Sigma, \Sigma', Q_0, \Delta)$, kde:

Q je konečná množina stavů,

Σ je konečná neprázdná množina vstupních symbolů (vstupní abeceda) taková, že $\Sigma \cap Q = \emptyset$,

Σ' je konečná neprázdná množina výstupních symbolů (výstupní abeceda), pro kterou platí $\Sigma' \cap Q = \emptyset$,

$Q_0 \subseteq Q$ je množina počátečních stavů,

Δ je konečná množina převodních pravidel. Podobně jako tomu bylo u stromového převodníku pracujícího zdola nahoru, také zde se můžeme setkat s dvěma různými typy pravidel. Prvním z nich je:

$$q(f(x_1, \ldots, x_n)) \to u[q_1(x_{i_1}), \ldots, q_k(x_{i_k})]$$

kde $f \in \Sigma_n$, $q, q_1, \ldots, q_k \in Q$, $x_1, \ldots, x_n, x_{i_1}, x_{i_k} \in \mathcal{X}_n$ a $u \in \mathcal{C}^k(\Sigma')$, a druhým typem pravidla, označovaným ε-pravidlo, je:

$$q(x) \to u[q_1(x), \ldots, q_k(x)]$$

kde $q, q_1, \ldots, q_k \in Q$, $x \in \mathcal{X}$ a $u \in \mathcal{C}^k(\Sigma')$.

Mějme dvojici termů $t, t' \in T(\Sigma \cup \Sigma' \cup Q)$. *Relace přechodu* $\vdash_\tau$ je definována následovně:

$$t \vdash_\tau t' \quad \overset{\text{def}}{\iff} \quad \begin{cases} \exists q(f(x_1, \ldots, x_n)) \to u[q_1(x'_1), \ldots, q_k(x'_k)] \in \Delta & \wedge \\ \exists C \in \mathcal{C}(\Sigma \cup \Sigma' \cup Q) & \wedge \\ \exists u_1, \ldots, u_n \in T(\Sigma) & \wedge \\ t = C[q(f(u_1, \ldots, u_n))] & \wedge \\ t' = C[u[q_1(v_1), \ldots, q_k(v_k)]], \text{ kde } v_i = u_j \text{ pokud } x'_i = x_j. \end{cases}$$

Reflexivně tranzitivní uzávěr relace $\vdash_\tau$ je $\vdash_\tau^*$. Relace R_τ indukovaná převodníkem τ je:

$$R_\tau = \left\{(t, t') \mid q(t) \vdash_\tau^* t' \wedge t \in T(\Sigma) \wedge t' \in T(\Sigma') \wedge q \in Q_0\right\}$$

Konečné stromové převodníky pracující shora dolů mohou nést některé z následujících přídomků – bez ε-pravidel, lineární, neodmazávající, úplný a deterministický – přičemž všechny tyto přídomky jsou definovány naprosto stejně jako u konečných stromových převodníků pracujících zdola nahoru (viz sekce 2.5.1).

2.5.3 Lineární stromové převodníky

Při aplikaci lineárního stromového převodníku τ na konečný stromový automat M, který reprezentuje regulární stromový jazyk $L(M)$, získáme nový konečný stromový automat M_τ reprezentující regulární stromový jazyk $L(M_\tau)$. Aplikace lineárního stromového převodníku vychází z teorému 2.9 a jejím důsledkem je ta skutečnost, že pro lineární stromový převodník lze vždy vyhodnotit výsledek aplikace transformace Υ_τ.

Teorém 2.9. (Transformace reprezentovaná lineárním stromovým převodníkem). *Definiční obor transformace Υ reprezentované stromovým převodníkem τ je regulární stromový jazyk. Je-li τ lineární, pak i obor hodnot Υ je regulárním stromovým jazykem.*

Pokud bysme potřebovali vyhodnotit zpětnou aplikaci stromového převodníku, nestačí nám pouze to, aby byl lineární. Menší opakování: stromový převodník je lineární, pokud jsou všechna jeho pravidla lineární, tj. na pravé straně pravidla smí být každá proměnná nejvýše jednou. Může se tedy stát, že aplikací lineárního stromového převodníku nahradíme určitý podterm jedinou konstantou. Obecně nelze tento krok provést zpětně, tzn. z termu získaného aplikací lineárního stromového převodníku nelze získat původní term.

2.5.4 Stromové převodníky zachovávající strukturu

Dvojice termů má *shodnou strukturu* právě tehdy, když jsou navzájem izomorfní, až na označení symbolů. Stromová transformace $\Upsilon : T(\Sigma) \times T(\Sigma')$ *zachovává strukturu* (v angličtině se nejčastěji používá pojem *shape preserving*) pouze tehdy, když $\forall(t, t') \in \Upsilon$ platí, že mají stejnou strukturu. *Stromový převodník zachovávající strukturu* τ je právě takový stromový převodník, jehož stromová transformace Υ_τ zachovává strukturu.

Tento problém řeší *stromové převodníky zachovávající strukturu*, což jsou stromové převodníky, které mohou v termu pouze zaměňovat symboly se stejnou aritou. Tyto stromové převodníky tedy povolují pouze pravidla tvaru:

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(g(x_1, , \ldots, x_n)$$

kde $n \geqslant 0$, $f \in \Sigma_n$, $g \in \Sigma'_n$ a $x_1, \ldots, x_n \in \mathcal{X}$. Alternativně lze tvar pravidla zapsat též ve zkráceném tvaru:

$$f(q_1, \ldots, q_n) \to q(g)$$

kde $\#(f) = \#(g) = n$, tzn. symboly f a g mají shodnou aritu. Je zřejmé, že každý stromový převodník zachovávající strukturu je lineární. V anglických textech se můžeme setkat s označením *Relabeling Tree Transducer*, které zachycuje podstatu této třídy stromových převodníků – stromový převodník zachovávající strukturu přepisuje v termu symboly abecedy Σ na symboly abecedy Σ'. Reprezentace převodní funkce stromového převodníku zachovávajícího strukturu je velice podobná reprezentaci přechodové funkce konečného stromového automatu. Převodní funkce Δ může být, stejně jako přechodová funkce konečných stromových automatů, reprezentována symbolicky, což umožňuje implementovat stromové převodníky zachovávající strukturu pomocí knihoven nástroje MONA. Stromové převodníky zachovávající strukturu lze implementovat jako konečný stromový automat nad abecedou $\Sigma \times \Sigma'$.

Příklad 2.8 zachycuje aplikaci triviálního stromového převodníku zachovávajícího strukturu na jeden term, přičemž výsledkem této aplikace je term se stejnou strukturou (stejným tvarem), ve kterém je symbol x zaměněn se symbolem a (ten býval jeho rodičem v původním termu).

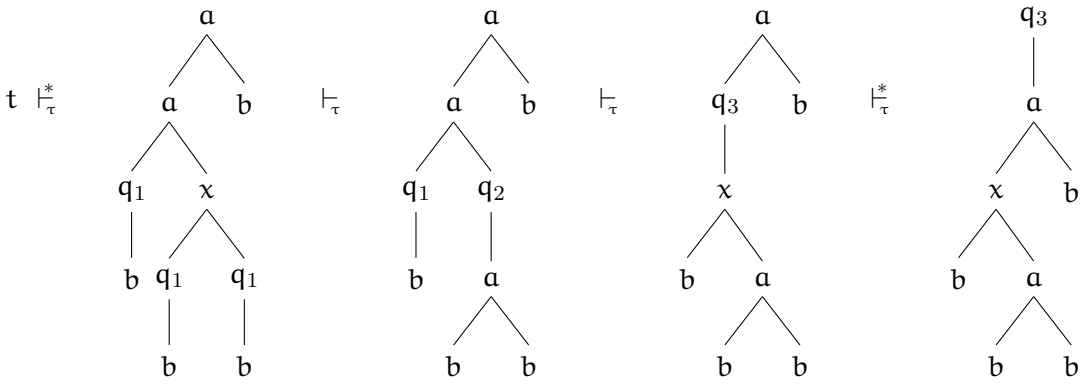
Mějme stromový převodník zachovávající strukturu τ. Převodník τ umožňuje vyhodnotit výsledek aplikace transformace Υ_τ a zároveň umožňuje vyhodnotit i výsledek zpětné aplikace, tj. aplikace transformace Υ_τ^{-1}.

Příklad 2.8 Konečný stromový převodník posouvající binární symbol x o jedno pozici výše (ukázka stromového převodníku zachovávajícího strukturu)

Mějme deterministický konečný stromový převodník pracující zdola nahoru $\tau = (Q, \Sigma, \Sigma', Q_f, \Delta)$, kde $\Sigma = \Sigma' = \{a_{(,)}, x_{(,)}, b_{()}\}$, $Q = \{q_1, q_2, q_3\}$, $Q_f = \{q_1, q_3\}$ a $\Delta =$

$$\left\{ \begin{array}{rclrcll} a(q_1,q_1) & \to & q_1(a), & x(q_1,q_1) & \to & q_2(a), & \\ a(q_1,q_2) & \to & q_3(x), & a(q_2,q_1) & \to & q_3(x), & \\ a(q_1,q_3) & \to & q_3(a), & a(q_3,q_1) & \to & q_3(a), & \\ b & \to & q_1(b) & & & & \end{array} \right\}.$$

Takto zadaný stromový převodník zachovává strukturu. Aplikaci τ, tzn. posuv symbolu x, lze ilustrovat pomocí následujícího běhu nad základním termem $t = a(a(b, x(b, b)), b)$:



Transformace definovaná stromovým převodníkem τ:

- nedovoluje posun více než jednoho binárního symbolu x;
- neumožňuje ztracení symbolu x z daného termu tím, že by byl posunut z kořene;
- umožňuje „prázdnou transformaci“ (platí pro termy, kde se nevyskytuje symbol x).

Pokud bysme výše uvedenou množinu přechodových pravidel doplnili o pravidlo $x(q_1, q_1) \to q_3(a)$, dosáhneme toho, že převodník se stane nedeterministickým.
Uvažujme tu možnost, že by τ nepracoval zdola nahoru (tj. po směru posuvu), ale shora dolů (tj. proti směru posuvu). V tomto případě by stromový převodník realizující danou transformaci byl nedeterministický. (Na třídě stromových převodníků nelze porovnat determinismus stromových převodníků pracujících zdola nahoru s determinismem stromových převodníků pracujících shora dolů.)
Za povšimnutí stojí také ta skutečnost, že pro stromové převodníky nemusejí být vstupní a výstupní abecedy navzájem disjunktní. V tomto příkladu jsou dokonce obě abecedy totožné.

Kapitola 3

Dostupné knihovny pro práci se stromovými automaty

V této kapitole jsou popsány vybrané implementace knihoven určených pro práci se stromovými automaty. Uvedeny jsou zde pouze knihovny použitelné pro formální analýzu a verifikaci: Timbuk (podkapitola 3.1) a MONA (podkapitola 3.2), přičemž druhý jmenovaný nástroj je rozebrán podrobně.

3.1 Timbuk

Timbuk [5] je sadou nástrojů určených pro dokazování dostupnosti nad přepisovacím systémem termů a pro manipulaci se stromovými automaty (nedeterministickými konečnými stromovými automaty pracujícími zdola nahoru). Tato knihovna je implementována v programovacím jazyce OCaml [6], populárním programovacím jazyce založeném na funkcionálním paradigmatu. V současné době jsou dostupné dvě primární verze softwaru Timbuk:

- Pro oblast formální verifikace a analýzy je zajímavá verze 2.2, jenž je stále dostupná na stránkách projektu.
- Poslední je verze 3.1, která však postrádá nástroje pro práci s konečnými stromovými automaty. Tato verze nástroje Timbuk se zaměřuje na analýzu dostupnosti a rovnicové aproximaci nad přepisovacími systémy termů.

Obě verze jsou zdarma dostupné ve zdrojové formě pod licencí GNU LGPLv2 [7].

3.2 MONA

MONA [1] je nástroj, jehož primární účel je rozhodování predikátů definovaných nad *redukovanou monadickou teorií druhého řádu jednoho následníka* (WS1S) a *redukovanou monadickou teorií druhého řádu dvou následníků* (WS2S), který je zdarma dostupný pod licencí GNU GPLv2 [8]. Predikáty nad WS1S lze interpretovat jako řetězce a predikáty nad WS2S lze interpretovat jako konečné stromy s přiřazenými stavy.

Na stránkách projektu [1] je v sekci „Publications“ (kromě manuálu [9]) dostupná celá řada článků, které se mj. zabývají koncepty a algoritmy, na nichž je nástroj MONA postaven [10, 11].

MONA využívá konečné automaty, resp. konečné stromové automaty, pro určení pravdivosti predikátů nad WS1S, resp. WS2S. Nástroj MONA využívá vlastní knihovny pro manipulaci s konečnými automaty a konečnými stromovými automaty, které jsou implementovány v jazyce C. Nejdůležitější vlastností, díky které lze nástroj MONA považovat za současný nejvýkonnější nástroj pro práci s konečnými stromovými automaty, je použití sdílených multiterminálních binárních rozhodovacích diagramů [12, 13] (tomuto tématu se více věnuje podkapitola 3.2.3) pro symbolickou reprezentaci přechodové funkce konečných stromových automatů i konečných automatů. Nástroj MONA dosahuje značné úspory potřebného paměťového prostoru, což umožňuje právě využití sdílených binárních rozhodovacích diagramů.

3.2.1 Reprezentace konečných stromových automatů v nástroji MONA

K rozhodování logiky WS2S jsou užívány konečné stromové automaty. Pro rozhodnutí predikátu nad WS2S a případné generování protipříkladu teoreticky postačuje konečný stromový automat pracující zdola nahoru, avšak při použití běžných stromových automatů pro rozhodování logiky WS2S se problematickým aspektem stává *stavová exploze*. Řešením, které používá nástroj MONA, je použití zvláštního typu stromových automatů, který se nazývá *Guided Tree Automaton*[*] (dále jen GTA) [10].

Před samotnou definicí GTA je nutné nadefinovat *guide*[*] $G = (D, \mu, d_0)$, kde:

D je konečná množina stavových prostorů,

$\mu : D \longmapsto D \times D$ je tzv. *guide function*[*],

$d_0 \in D$ je název počátečního stavového prostorů.

Guide je deterministický konečný stromový automat pracující shora dolů, který zcela ignoruje stavy přiřazené podtermům. Stavy automatu G jsou následně používané pro určení stavového prostoru pro konečné stromové automaty pracující zdola nahoru.

GTA M_G definovaný pro guide G je množina deterministických konečných stromových automatů pracujících zdola nahoru: $M_G = (\{Q_d\}, \Sigma, \{\delta_d\}, \{q^0_d\}, Q_f)$, kde:

$\{Q_d\}$ je množina navzájem disjunktních konečných množin stavů (právě jedna pro každý stavový prostor $d \in D$),

Σ je ohodnocená abeceda,

$\{\delta_d\}$ je množina funkcí přechodu (právě jedna pro každý stavový prostor $d \in D$) taková, že pro $\mu(d) = (d_1, d_2)$ je δ_d tvaru $\delta_d : (Q_{d_1} \times Q_{d_2}) \longmapsto (\Sigma \longmapsto Q_d)$,

$\{q^0_d\}$ je množina počátečních stavů taková, že $\forall d \in D : q^0_d \in Q_d$,

$Q_f \in Q_{d_0}$ je množina koncových stavů.

Rozhodování o tom, zdali GTA M_G přijímá základní term t je proces, který se skládá ze dvou fází:

[*]Tento pojem zde nebude překládán.

1. Nejprve je průchodem shora dolů přiřazen stavový prostor každému uzlu daného termu. Pro tento účel slouží funkce $\hat{\mu} : T(\Sigma) \times D \longmapsto T(\Sigma, D)$, pro kterou platí:

$$\hat{\mu}(t^0, d) = t^0_d$$
$$\hat{\mu}(t(t', t''), d) = t_d(\hat{\mu}(t', d_1), \hat{\mu}(t'', d_2))$$

   kde $\mu(d) = (d_1, d_2)$, $t^0, t, t', t'' \in T(\Sigma)$, $t' = t|_1$, $t'' = t|_2$ a $\|t^0\| = 1$ (nemá žádné podtermy). Zápis t_d znamená, že kořenu termu t je přiřazen stavový prostor d.

2. V dalším kroku jsou pro každý podstrom kořenového stavového prostoru $\hat{\mu}(t, d_0)$ přiřazovány stavy. Samotné přiřazení stavu je prováděno zdola nahoru pomocí funkce: $\hat{\delta} : T(\Sigma, D) \longmapsto \bigcup_{d \in D} Q_d$ pro kterou platí:

$$\hat{\delta}(t^0_d) = \delta_d(q^0_d, q^0_d)(Head(t^0))$$
$$\hat{\delta}(t_d(t', t'')) = \delta_d(\hat{\delta}(t'), \hat{\delta}(t''))(Head(t))$$

   kde $t^0, t, t', t'' \in T(\Sigma)$, $t' = t|_1$, $t'' = t|_2$, $\|t^0\| = 1$ a q^0_d je počáteční stav daného stavového prostoru d.

Jazyk přijímaný pomocí M_G je množinou stromů $t \in T(\Sigma)$ takových, že $\hat{\delta} \circ \hat{\mu}(t, d_0) \in Q_f$. Na GTA můžeme nahlížet jako na deterministický konečný stromový automat pracující zdola nahoru, který pomocí G nejprve přidělí stavový prostor jednotlivým uzlům stromu t. Pokud by existoval GTA s jediným stavovým prostorem (což nástroj MONA neumožňuje) jednalo by se o deterministický konečný stromový automat pracující zdola nahoru, který pracuje nad základními termy skládajícími se pouze z nulárních až binárních symbolů.

3.2.2 Universa a stavové prostory

Universum u je takový podstrom nekonečného binárního stromu t, že pro libovolnou pozici p z universa u platí, že levý i pravý následník p patří také u. Universum lze jednoznačně identifikovat cestou, tzn. řetězcem 0 a 1 reprezentujících levého a pravého následníka, od kořene stromu t ke kořenu daného universa. K universu u je přiřazena neprázdná množina D_u, která obsahuje všechny stavové prostory dosažitelné z kořene u. Stavový prostor $d \in D_u$ právě tehdy, když u obsahuje uzel, kterému je přiřazen stavový prostor d.

Vzhledem k tomu, že nástroj MONA pracuje s tzv. booleovským stavovým prostorem, který musí být oddělen od stavových prostorů přidělených universům, musí každý GTA definovat přinejmenším dvě universa. Booleovský stavový prostor je rezervován pro interpretaci proměnných typu boolean.

Nástroj MONA umožňuje čtyři různé přístupy, jak lze definovat universa:

1. Není definován guide ani universa. Nástroj MONA vygeneruje triviální guide (viz příklad 3.2) a dvě universa, přičemž pouze jedno universum je využito. (Kvůli booleovskému stavovému prostoru jsou potřeba definovat alespoň dvě universa.)

2. Není definován guide, avšak uživatel specifikoval universa. Nástroj MONA v tomto případě vygeneruje guide jako vyvážený strom a jednotlivá universa umístí do listových uzlů tohoto stromu tak, aby každé universum mělo přiděleno právě jeden stavový prostor. Pouze v případě, když je explicitně deklarováno jediné universum, je implicitně přidáno universum dummy.

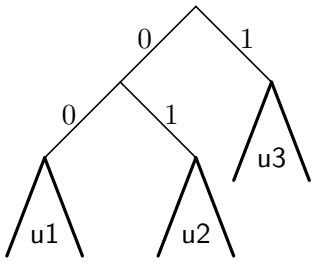
3. Uživatel kompletně definoval guide i všechna potřebná universa.

Příklad 3.1 Rozdělení binárního stromu na universa (převzato z [9])

Mějme guide, který je definovaný pomocí následující konstrukce guide:

```
guide a->(b,c), b->(d,e), c->(c,c), d->(d,d), e->(e,f), f->(f,f);
```

Tento guide pracuje se stavovými prostory označenými a až f, přičemž první uvedený stavový prostor, tj. a, je vyhrazen pro tzv. *booleovský stavový prostor* (viz dále). Za klíčovým slovem guide jsou vyjmenována všechna pravidla popisující guide function – funkci pokrývající GTA stavovými prostory. Tímto zápisem rozdělíme nekonečný binární strom na tři universa, u1, u2 a u3, která lze identifikovat řetězci 00, 01 a 1. Toto rozdělení lze graficky reprezentovat např. takto:



Kořen stromu a jeho levý následník nepatří do žádného universa, kdežto všechny ostatní uzly patří do právě jednoho universa. Přiřazení stavových prostorů do univers je následující: $D_{u1} = \{d\}, D_{u2} = \{e, f\}, D_{u3} = \{c\}$. Nástroj MONA umožňuje uživateli, aby tato universa definoval pomocí následující deklarace:

```
universe u1:00, u2:01, u3:1;
```

Pozn.: Ukázky zdrojového kódu v rámci tohoto příkladu jsou segmenty vstupního souboru pro nástroj MONA, tzn. definují guide a universa pro GTA rozhodující určitý predikát nad WS2S.

4. Uživatel definoval alespoň jeden rekurzivní typ. (Tato problematika přesahuje rámec této práce. Případným zájemcům doporučuji nastudovat sekci 7.9 publikace [9].)

Nástroj MONA se snaží zabránit stavové explozi tím, že používá GTA, který přechodovou funkci konečného stromového automatu rozděluje na několik částí, přičemž každá část přechodové funkce je definována nad právě jedním universem.

3.2.3 Multiterminální binární rozhodovací diagramy

Multiterminální binární rozhodovací diagramy (MTBDD) [13] jsou rozšířením *redukovaných uspořádaných binárních rozhodovacích diagramů* (ROBDD). Akronym ROBDD bývá často zkracován jako BDD, a tak se s akronymem ROBDD setkáváme pouze v případech, kdy chce autor zdůraznit aspekty redukce a uspořádání. Primárním zdrojem informací o BDD mi byla kapitola 7.1.4 v [13]. V rámci této diplomové práce není nutné explicitně zdůrazňovat tyto vlastnosti, a tak budeme akronymem BDD označovat redukovaný uspořádaný binární rozhodovací diagram.

Binární rozhodovací diagram je datová struktura využívaná pro reprezentaci a manipulaci s Booleovskými funkcemi ve světě počítačů. BDD tedy reprezentuje funkci $f : \{0,1\}^n \longmapsto \{0,1\}$. Binární rozhodovací diagram můžeme získat aplikací určitých redukč-

Příklad 3.2 Triviální guide

Triviální guide je nejmenší guide, který lze použít při práci s nástrojem MONA nebo jeho knihovnami pro práci se stromovými automaty. Zdrojový kód pro triviální guide je:

```
guide:
   <hat> 1 2
   <univ> 1 1
   <dummy> 2 2
```

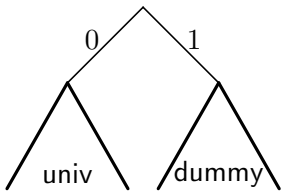
Triviální guide se skládá ze třech stavových prostorů:

- hat je booleovský stavový prostor, který nepatří do žádného universa;
- univ je „hlavní“ stavový prostor, kde je umístěn multiterminální binární rozhodovací diagram, který reprezentuje přechodovou funkci konečného stromového automatu pracujícího zdola nahoru, definovaný uživatelem;
- dummy se nechává prázdný.

Pro tuto trojici stavových prostorů se definují pouze dvě universa, protože booleovský stavový prostor nesmí být umístěn v žádném universu. Ve zdrojovém kódu se každému názvu universa pouze přiřadí jeho jedinečný číselný identifikátor:

```
universes:
   <univ> 0
   <dummy> 1
```

Pokud použijeme triviální guide, získáme následující rozdělení na universa:



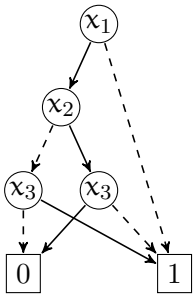
Pozn.: Ukázky zdrojového kódu v rámci tohoto příkladu jsou segmenty vstupního souboru pro knihovnu nástroje MONA, která slouží pro práci se stromovými automaty (konkrétně s GTA).

ních kroků na binární rozhodovací strom. Příklad 3.3 znázorňuje binární rozhodovací diagram jednoduché Booleovské funkce o třech proměnných: x_1, x_2 a x_3. Nejvýše postavený uzel slouží jako vstupní bod do daného BDD, označujeme jej kořen. Každému nelistovému uzlu je přiřazena právě jedna proměnná, např. kořen v příkladu 3.3 má přiřazenou proměnnou x_1. Každý nelistový uzel má právě dva následníky, kteří jsou v binárním rozhodovacím diagramu naznačeny pomocí orientovaných hran. K nízkému z následníků vede čárkovaná hrana, kterou označujeme 0 (nebo též LO); k vysokému následníkovi vede plná hrana, jenž označujeme 1 (nebo též HI). Přes nelistový uzel vede cesta pro libovolné ohodnocení dané proměnné: pro uzel ohodnocený proměnnou x_i pokračujeme na nízkého následníka, pokud platí $x_i = 0$, nebo pokračujeme na vysokého následníka, pokud platí $x_i = 1$. Jakmile se při průchodu binárním rozhodovacím diagramem dostaneme do listového uzlu označeného 0, resp. 1, je Booleovská funkce f pro dané ohodnocení proměnných vyhodnocena jako nepravdivá, resp. pravdivá.

BDD má pouze dvojici listových uzlů, tj. 0 a 1, což může být v mnoha případech příliš omezující. Oproti tomu MTBDD může mít i více než dva listové uzly, díky čemuž lze

Příklad 3.3 Binární rozhodovací diagram

BDD reprezentující Booleovskou funkci $x_1x_2\neg x_3 + x_1\neg x_2x_3 + \neg x_1$ je znázorněn zde:

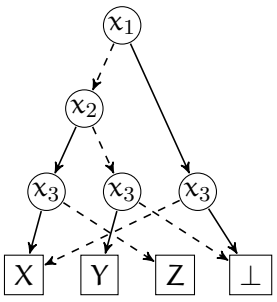


MTBDD využít pro symbolickou reprezentaci přechodové funkce GTA (při použití BDD by byl GTA omezen na pouhou dvojici stavů). Ukázka MTBDD je uvedena v příkladu 3.4. MTBDD již nereprezentuje funkci f, namísto toho reprezentuje funkci $g : \{0,1\}^n \longmapsto \mathbb{D}$, kde $\mathbb{D}$ je libovolná množina taková, že $\bot \in \mathbb{D}$.

Při použití MTBDD pro reprezentaci přechodové funkce konečného stromového automatu, je každý symbol abecedy Σ zakódován ve formě binárního řetězce, tzn. existuje kódovací funkce $e : \Sigma \longmapsto \{0,1\}^n$, kde $n \in \mathbb{N}$ takové, že $n \geqslant \lceil \log |\Sigma| \rceil$. Pokud je každý vstupní symbol $a \in \Sigma$ zakódován pomocí funkce e, pak platí $e(a) = (x_1, \ldots, x_n)$, kde $x_1, \ldots, x_n$ jsou odpovídající proměnné MTBDD. Pokud jsou listovým uzlům přiřazeny stavy konečného stromového automatu, reprezentuje MTBDD přechodovou funkci konečného stromového automatu z jednoho stavu. Pro každý symbol a pak existuje právě jeden možný průchod MTBDD, kterým se dostane do listového uzlu s ohodnocením odpovídajícím následujícímu stavu automatu.

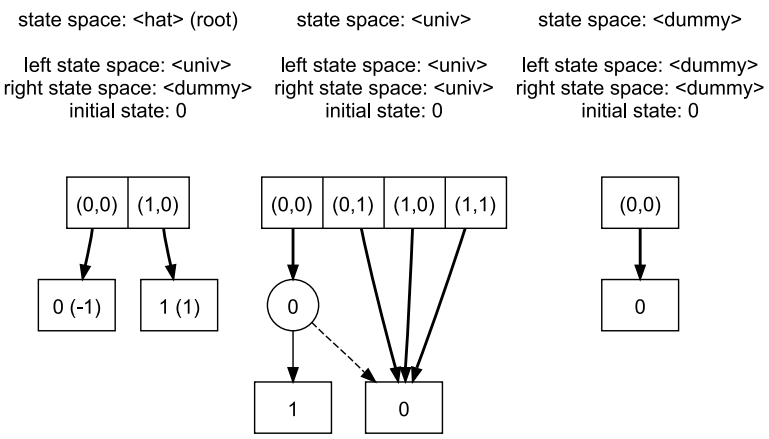
Příklad 3.4 Multiterminální binární rozhodovací diagram

MTBDD reprezentující formuli $x_1\neg x_3X + \neg x_1x_2x_3X + \neg x_1\neg x_2x_3Y + \neg x_1x_2\neg x_3Z$ je znázorněn zde:



Na výše uvedené ilustraci je znázorněn MTBDD se čtyřmi terminálními uzly a $\bot$. Terminální uzly X, Y a Z reprezentují možné výsledky dané formule, které můžeme interpretovat jako pravdivé (zadaná formule má právě tři odlišné pravdivé výsledky). Oproti tomu terminální uzel $\bot$ reprezentuje všechna nepravdivá ohodnocení dané formule. Můžeme si všimnout jisté analogie mezi „sink stavem“ konečného automatu a terminálním uzlem $\bot$.

Obrázek 3.1 Sdílené MTBDD v nástroji MONA (používá triviální guide)



Každá dvojice stavů konečného stromového automatu může mít vlastní MTBDD nebo se použije *sdílený* MTBDD. Tímto rozšířením se spojí všechny všechny diagramy do jednoho *orientovaného acyklického grafu* (DAG), který obsahuje jeden kořen pro každou dvojici stavů konečného stromového automatu. Důležité je uvědomit si, že nástroj MONA pracuje s GTA, který využívá stavových prostorů. Každý stavový prostor pracuje s vlastní množinou stavů a vlastní přechodovou funkcí, tzn. každý stavový prostor má pro reprezentaci přechodové funkce vyhrazen svůj vlastní sdílený MTBDD.

3.3 Implementační detaily knihoven nástroje MONA

Pro práci s konečnými stromovými automaty lze využívat pouze GTA knihovny, nebo je možné využít i BDD knihovny, která poskytuje přístup k samotné symbolické reprezentaci přechodové funkce konečných stromových automatů. V obou případech je vhodné znát některé implementační detaily, které jsou shrnuty v následujících sekcích.

3.3.1 Implementace BDD knihovny

Při implementaci knihovny pro práci s BDD byl kladen důraz na co nejvyšší rychlost operací nad BDD. Jeden uzel BDD je popsán strukturou bdd_record, která je v paměti uložena na 16 B. Autoři knihovny se snažili záměrně udržet velikost této struktury malou, aby bylo možné současně nahrát dvě tyto struktury do cache 32b procesoru. Nejpodstatnější složkou struktury bdd_record je pole dvou prvků typu unsigned pojmenované lri[2], které zabírá rovnou polovinu celkové velikosti struktury bdd_record, tj. 8 B. Jak samotná název napovídá, lri v sobě nese následující trojici hodnot:

- *Index bitu ve vstupním symbolu* je uložen na 2 B. Když vezmeme v potaz, že hodnota $2^{16} - 1$ je rezervována pro označení listových uzlů BDD, lze indexovat až $2^{16} - 1 = 65\,535$ proměnných (tzn. bitů ve vstupním symbolu).
- *Odkaz na nízkého následníka* je uložen na 3 B. Z velikosti odkazu lze odvodit ta skutečnost, že sdílený MTBDD může obsahovat maximálně $2^{24} = 16\,777\,216$ uzlů, které v paměti zaberou 256 MB.

- Poslední hodnotou je *odkaz na vysokého následníka*, která je také uložena na 3 B.

BDD uzly jsou v paměti ukládány do hashovací tabulky, a tak je přístup k BDD uzlům relativně rychlý oproti použití jiných datových struktur (např. jednosměrně a dvojsměrně lineárně vázaných seznamů). Hashovací tabulka používá index-sekvenčního přístupu umožňujícího rychlý přístup k BDD uzlům, avšak rychlost přístupu je závislá na obsazenosti tabulky. Čím více BDD uzlů je v hashovací tabulce uloženo, tím je vyšší průměrná přístupová doba do této tabulky. Kvůli tomu je při dosažení kritické hranice obsazenosti nutné zvětšit kapacitu hashovací tabulky. Nástroj MONA zvětšuje kapacitu hashovací tabulky pro uchování BDD uzlů na dvojnásobek původní kapacity, a tak dojde k přemístění BDD uzlů v rámci hashovací tabulky.

Pro uchovávání údajů o celém BDD slouží struktura pojmenovaná bdd_manager. Tato struktura má na starosti především hashovací tabulku, v níž jsou uchovány veškeré BDD uzly, a cache paměť pro uchovávání dříve vyhodnocených výsledků.

Je nutné podotknout, že odkazy na oba následníky (nízkého a vysokého) BDD uzlu jsou datového typu bdd_ptr, což je přímý odkaz do hashovací tabulky. Zřejmě lze hodnoty typu bdd_ptr považovat za volatilní data, protože při zdvojnásobení kapacity hashovací tabulky přestávají být tyto hodnoty platné. Pokud programátor potřebuje držet perzistentní odkaz na BDD uzel, poskytuje mu bdd_manager alternativní způsob pro uchovávání výsledků BDD operací, kterým je pole bdd_roots, které je přepočítáno při zdvojnásobení velikosti hashovací tabulky. Index do pole bdd_roots je datového typu bdd_handle. Struktura bdd_manager při zdvojnásobení kapacity hashovací tabulky automaticky aktualizuje hodnoty v poli bdd_roots. Pro práci s indexem typu bdd_handle je přepraveno několik maker:

- BDD_ADD_ROOT přidá jeden volatilní odkaz bdd_ptr na BDD uzel do pole bdd_roots.
- BDD_LAST_HANDLE vrací index do pole bdd_roots, kam byl naposledy uložen volatilní odkaz.
- BDD_ADD_ROOT_SET_HANDLE je spojením předchozích dvou maker. Vloží volatilní odkaz do pole bdd_roots a zároveň vrátí index pozice, na kterou byl uložen.
- BDD_ROOT vrací v současné době platný volatilní odkaz na BDD uzel, který leží na požadovaném indexu v poli bdd_roots.

V této podkapitole byly vysvětleny základní principy, na kterých ke postavena implementace BDD knihovny nástroje MONA. Informace o algoritmech používaných pro práci s BDD jsou uvedeny [12], avšak dané téma je již nad rámec této zprávy.

3.3.2 Implementace GTA knihovny

V kapitole 3.2.1 jsou uvedeny formální definice konečného stromového automatu (dále jen GTA), který je implementován v rámci GTA knihovny nástroje MONA. GTA pracuje pouze nad binárními termy, tzn. každý podterm má v kořenu binární symbol nebo nulární symbol. Konstanty jsou v nástroji MONA zastoupeny tzv. počátečním stavem $q_d^0 \in Q_d$ (pro každý stavový prostor d je definován právě jeden) tak, že místo pravidla tvaru $a \to q$ má konečný stromový automat pravidlo tvaru $\left(a(q_d^0, q_d^0) \to q\right) \in \delta_d$, kde $a \in \Sigma_0$ a $q \in Q_d$. Jinak řečeno, nástroj MONA nerozlišuje mezi binárními symboly a konstantami.

Tato knihovna poskytuje sadu funkcí, které umožňují relativně jednoduchým způsobem implementovat konstrukci GTA přechodové funkce GTA. Tyto funkce aplikují spoustu operací nad sdíleným MTBDD, který je postupně vytvářen. Způsob použití těchto funkcí při

konstrukci GTA je znázorněn v pseudokódu uvedeném v rámci algoritmu 1 a zde následuje jejich popis:

- void gtaSetup (unsigned počet_stavů); – první krok, který provede přípravu na vytvoření GTA, který bude mít kořenovém stavovém prostoru počet_stavů stavů.
- void gtaSetupDelta (SsId d, unsigned velikost_l, unsigned velikost_p, unsigned *indexy, unsigned počet_indexů);
  provede přípravu pro konstrukci přechodů ve stavovém prostoru d s levým, resp. pravým, stavovým prostorem d obsahujícím právě velikost_l, resp. velikost_p, stavů. Pole indexy obsahuje právě počet_indexů proměnných. (Jedná se o proměnné, která načetl nástroj MONA ze zadaného predikátu nad logikou WS2S – pro naše potřeby je tato hodnota nepodstatná.)
- void gtaAllocExceptions (SsId l, SsId p, unsigned počet_cest); – definuje počet cest sdíleným MTBDD, které vedou ze stavu (l, p) do jiného než implicitního stavu. Je nutné definovat přesně počet_cest těchto cest do explicitně udaného stavu, což se provede opakovaným voláním následující funkce.
- void gtaStoreException (unsigned s, char *cesta); – uloží jednu z cest, které vedou do jiného než implicitního stavu: řetězec cesta je bitový řetězec popisující volbu cesty při průchodu sdíleným MTBDD a s je stav, do kterého tato cesta vede.
- void gtaStoreDefault (unsigned s); – nastaví implicitní stav na s, tzn. všechny cesty, které nebyly explicitně definovány pomocí předchozí funkce vedou do stavu s.
- void gtaBuildDelta (unsigned s); – nastaví počáteční stav na s.
- GTA *gtaBuild (char konečné_stavy[]); – zkonstruuje přechodovou funkci konečného stromového automatu, která byla vytvořena předchozími funkcemi, přičemž řetězec konečné_stavy obsahuje právě počet znaků, kde + definuje odpovídající stav za koncový stav a - definuje odpovídající stav za nekoncový stav. (Pokud řetězec konečné_stavy obsahuji i jiné znaky, pak jsou stavy s daným indexem považovány za tzv. *don't care*, které jsou pro naše potřeby nezajímavé.)

Algoritmus 1 Použití funkcí z GTA knihovny pro konstrukci přechodové funkce konečného stromového automatu (převzato z [9])

```
CONSTRUCT-DELTA–NOT-USED()
 1  gtaSetup(počet stavů v kořenovém stavovém prostoru)
 2  for ∀d∈D
 3      do gtaSetupDelta(d, velikost levého stav. prost., velikost pravého stav. prost.,
 4                       pole indexů, počet indexů)
 5         for ∀(l,p)∈Q_d × Q_d
 6             do gtaAllocExceptions(l, p, počet cest)
 7                for ∀i∈{1,...,počet cest}
 8                    do gtaStoreException(cílový stav cesty, cesta)
 9                gtaStoreDefault(implicitní stav)
10         gtaBuildDelta(q_d^0)
11  return gtaBuild(pole určující konečné stavy)
```

Jak již bylo řečeno dříve, nástroj MONA není primárně určen pro práci s konečnými stromovými automaty, ale pro vyhodnocování pravdivosti predikátů nad logikou WS2S, z čehož vyplývá také odlišný přístup k reprezentaci symbolů. Symboly jsou v knihovnách nástroje MONA kódovány do řetězce bitů. Vyhodnocování dalšího stavu GTA by probíhalo takovýmto způsobem:

1. Pomocí současného stavu reprezentovaného uspořádanou dvojicí (l, p) naindexuje tabulku kořenů, čímž se dostane na první BDD uzel.

2. Podívá se na první položku BDD uzlu, tj. index bitu ve vstupním symbolu, a podle jeho hodnoty provede jeden z následujících kroků:

   3.a Je-li index bitu ve vstupním symbolu roven -1, pak dorazil do listového BDD uzlu, tzn. našel další stav GTA a končí. Index nového stavu je načten z položky BDD uzlu, která jinak slouží pro uchování odkazu na nízkého potomka (taková položka u listového BDD uzlu zřejmě nedává smysl).

   3.b Je-li index bitu rozdílný od -1, pak stale ještě nedošel do listového BDD uzlu. Podívá se do vstupního symbolu na daný index, čímž získá hodnotu 0 nebo 1. Pokud našel hodnotu 0, pak se přesune na nízkého následníka. V opačném případě se přesune na vysokého následníka. Pokračuje bodem 2.

Při vyhodnocování dalšího stavu na základě vstupního symbolu je procházen MTBDD, přičemž každý BDD uzel je definován trojicí hodnot: indexem do bitového řetězce symbolu, a dvojici ukazatelů na další BDD uzel.

3.3.3 Implementace knihovny pro správu paměti

Nástroj MONA a jeho knihovny využívají vlastní knihovnu pro správu paměti, která mimo jiné implementuje vlastní verzi funkce pro alokaci paměti. V některých případech, jako je např. kompilace a sestavení nástroje MONA nebo jeho knihoven na 64b architektuře, nemusí dojít k úspěšné kompilaci nebo bude výsledný software chybně pracovat s pamětí. V takovém případě je nutné přidat při kompilaci volbu -DUSE_MALLOC. Tuto volbu je mj. nutné použít při propojení knihoven nástroje MONA s dalšími programovacími jazyky.

Kapitola 4

Současný formát vstupu a výstupu knihoven MONA

Nástroj MONA není primárně určen pro práci s konečnými stromovými automaty, avšak poskytuje pro tyto účely kvalitně zpracovanou knihovnu. Hlavní nevýhodou této knihovny je fakt, že formát, v jakém vyžaduje vstupní soubory je relativně náročný na pochopení a vytvoření vstupu ve správném formátu zabere skutečně velké množství času.

Ukázkový vstupní soubor je uveden v příkladu 4.1. Díky tomu, že vstupní soubor je uložen v textové podobě, je zaručena určitá úroveň čitelnosti, avšak produkování podobného souboru je velmi náročné – je nutné brát v potaz tu skutečnost, že uvedený ukázkový soubor reprezentuje triviální konečný stromový automat, což se odráží hlavně na sekci bdd, která bývá u reálných vstupních souborů o několik řádů větší. Následuje podrobný rozbor ukázkového vstupního souboru:

1 MONA GTA
Hlavička souboru informuje o tom, že tento soubor definuje GTA (konečný stromový automat) pro nástroj MONA.

2 number of variables: 0
Udává *počet proměnných* logiky WS2S. V tomto případě se jedná o konečný stromový automat, který není používán pro vyhodnocení určitého predikátu definovaného nad WS2S, a tak nemá definovány žádné proměnné.

3 state spaces: 3
Tento údaj určuje *počet stavových prostorů* daného GTA. Tento GTA používá triviální guide, a tak využívá právě tři stavové prostory (nejmenší počet, jaký nástroj MONA dovoluje).

4 universes: 2
Počet univers daného GTA. Podobně jako u předchozího řádku, i zde je zřejmý udávaný počet univers.

5–6 state space sizes: 2 2 1
final: -1 1
Řádek 5 udává velikost stavových prostorů, tj. počet stavů v každém z definovaných stavových prostorů. Pořadí stavových prostorů odpovídá pořadí, v jakém jsou stavové prostory definovány v klauzuli guide. Na řádku 6 jsou uvedeny třídy, do nichž spadají

Příklad 4.1 Vstupní soubor knihovny pro práci s konečnými stromovými automaty

```
1   MONA GTA
2   number of variables: 0
3   state spaces: 3
4   universes: 2
5   state space sizes: 2 2 1
6   final: -1 1
7   guide:
8      <hat> 1 2
9      <univ> 1 1
10     <dummy> 2 2
11  types: 0
12  universes:
13     <univ> 0
14     <dummy> 1
15  variable orders and state spaces:
16
17  state space 0:
18     initial state: 0
19     bdd nodes: 2
20     behaviour:
21        0
22        1
23     bdd:
24        -1 0 0
25        -1 1 0
26
27  state space 1:
28  initial state: 0
29     bdd nodes: 3
30     behaviour:
31        0 1
32        1 1
33     bdd:
34        4 1 2
35        -1 0 0
36        -1 1 0
37
38  state space 2:
39     initial state: 0
40     bdd nodes: 1
41     behaviour:
42        0
43     bdd:
44        -1 0 0
45
46  end
```

stavy definované na předchozím řádku: Stav 0 (definován pro všechny tři stavové prostory) spadá do třídy -1 a stav 1 (definován pouze pro první dva stavy) spadá do třídy 1.

Nástroj MONA rozlišuje tři různé třídy stavů:

- $-\mathbf{1}$ je třída *přijímajících stavů*,
- **1** je třída *odmítajících stavů*
- a třída **0** je označována jako *don't care* (čili stavy, které nás nezajímají).

7–10 guide:
<hat> 1 2
<univ> 1 1
<dummy> 2 2
Tyto řádky popisují *guide*, který určuje stavové prostory (viz příklad 3.2).

11 types: 0
Tato hodnota je využívána pouze při práci s logikou WS2S.

12–14 universes:
<univ> 0
<dummy> 1
V této části jsou *definována universa*. Za identifikátorem universa se vždy udává posloupnost hran, kterými se guide do daného universa dostane – odtud se doplní informace o tom, které stavové prostory jsou přiřazeny do daného universa. Způsob, jakým je tento GTA rozdělen na universa, je uveden v příkladu 3.2.

15 variable orders and state spaces:
Tato klauzule se využívá pro *definici proměnných* logiky WS2S. V tomto případě je prázdná (souvisí s počtem pravidel, viz řádek 2). Běžně obsahuje prvky tvaru P 2: 1, kde P je identifikátor dané proměnné, 2 značí řád proměnné (platí $\mathsf{P} \in \{0, 1, 2\}$) a 1 udává stavový prostor, do kterého je daná proměnná přiřazena (1 by v tomto případě označovala <univ>).

17–25 state space 0: ...
Definice nultého stavového prostoru, tj. <hat>. Popis definice stavového prostoru je uveden v dalším bodu.

27–36 state space 1: ...
Definice stavového prostoru s pořadím 1, tj. <univ>. Každá definice stavového prostoru se skládá z následujících podčástí:

- initial state: 0
  Tento řádek udává *počáteční stav* konečného stromového automatu v daném stavovém prostoru.
- bdd nodes: 3
  Udává počet uzlů (včetně listových uzlů) sdíleného MTBDD, tj. binárního rozhodovacího diagramy reprezentujícího přechodovou funkci pro daný stavový prostor.
- behaviour:
  Následuje matice obsahující $n \times m$ indexů, kde n udává velikost stavového prostoru levého následníka (počet řádků matice) a m udává velikost stavového pro-

storu pravého následníka (počet sloupců matice). Pomocí matice behaviour se indexují kořeny (vstupní body) sdíleného MTBDD. Vzhledem k tomu, že v každém stavovém prostoru je samostatný DFTA pracující zdola nahoru nad binárními stromy, je zřejmé, že pro určení kořene (z něhož se bude procházet BDD) jsou nutné informace od obou následníků. Při načtení konstanty $a \in \Sigma_0$ bude v tomto případě používán kořen jehož index je v matici behaviour umístěn na pozici (0, 0), což jsou hodnoty definované jako initial state tohoto stavového prostoru.

– bdd:
V této části textově vypsána reprezentace sdílené MTBDD, které se více věnuje kapitola 3.3.1.

38–44 state space 2: ...
Definice stavového prostoru s pořadím 2, tj. <dummy>. Tento stavový prostor je zcela nevyužit, avšak kvůli booleovskému stavovému prostoru (tj. <hat>) je jeho definice nutná.

46 end
Informace o dosažení konce důležitých údajů ve vstupním souboru.

Konečný stromový automat načtený ze vstupního souboru lze převést do grafické reprezentace pomocí nástroje gta2dot, jehož výstupem je soubor formátu .dot. Pro vygenerování grafické reprezentace ze souboru ve formátu .dot lze použít nástroj dot, který je zdarma (pod licencí EPLv1 [14]) dostupný v rámci softwarového balíčku graphviz [15]. Nástroj dot poskytuje celou řadu výstupních formátů, např. svg, png, gif, jpeg. Grafická reprezentace konečného stromového automatu uvedeného v příkladu 4.1 je znázorněna na obr. 3.1.

Kapitola 5

Návrh implementace

V kapitole 4 byl uveden podrobný popis formátu vstupu a výstupu, který v současné době využívá GTA knihovna nástroje MONA.

V rámti této kapitoly bude věnován prostor návrhu nového formátu vstupu a výstupu GTA knihovny nástroje MONA, které jsou v souladu s matematickými definicemi konečných stromových automatů (podkapitola 5.1). Zde bude věnován prostor navrhovaným formátům souborů pro uložení definic abeced, konečných stromových automatů a stromových převodníků. Dále je uveden konceptuální návrh datových struktur (podkapitola 5.2). Posléze je věnován prostor návrhu interaktivního rozhraní pro manipulaci s konečnými stromovými automaty a stromovými převodníky, pro jehož implementaci byl zvolen jazyk Haskell [16] (podkapitola 5.3). Závěr kapitoly je věnován krátkému přehledu požadavků kladeným na implementovaná rozhraní (podkapitola 5.4).

5.1 Návrh nového formátu vstupu a výstupu

Nově navrhovaný formát vstupů pro GTA knihovnu nástroje MONA má přinést zásadní zjednodušení, které by usnadnilo práci s konečnými stromovými automaty. Mezi nejvíce zásadní změny patří zavedení abecedy a použití pravidel pro popis přechodové funkce konečného stromového automatu. V době návrhu byla hlavním zdrojem inspirace knihovna z nástroje Timbuk [5] ve verzi 2.2 (viz sekce 3.1). Všechny definice jsou uloženy ve formě textových ASCII souborů.

5.1.1 Definice abecedy

Zavedení abecedy je poměrně zásadní změnou, která umožní zjednodušit zápis pravidel reprezentujících přechodovou funkci automatu. Uživatel bude mít možnost vytvořit soubor s definicí abecedy vlastnoručně, nebo bude možné využít jednoduchého nástroje pro vygenerování souboru obsahujícího definici abecedy.

Abeceda je vždy umístěna v samostatném souboru, který obsahuje definice jednotlivých symbolů abecedy. Pro definici symbolu abecedy může uživatel využít jeden z následujících dvou zápisů:

Definice symbolu s implicitním odvozením bitového řetězce: `a`,
kde `a` je identifikátor označující daný symbol, který musí být v rámci abecedy jedinečný, tzn. nelze definovat dva symboly se stejným identifikátorem. Délka identifikátoru symbolu je omezena konstantou IDENTIFIER_LENGTH, která může být nastavena

při překladu daného modulu. Identifikátor smí obsahovat pouze malé a velké znaky anglické abecedy, číslice a podtržítko.

Definice symbolu s explicitním uvedením bitového řetězce: $a(x_1 \dots x_k)$,
kde $x_1 \dots x_k$ je bitový řetězec reprezentující zakódovaný symbol a, pro který musí platit $\forall i \in \{1, \dots, k\} : x_i \in \{0, 1, X\}$, kde 0 a 1 jsou hodnoty binární číslice a X je zástupný znak. Pomocí zástupného znaku X uživatel dá najevo to, že hodnota bitu na dané pozici je ignorována.

Uživatel může při vytváření definice abecedy kombinovat oba výše uvedené způsoby definice symbolů. Teprve po načtení celé definice abecedy budou generovány bitové řetězce pro symboly s implicitním odvozením bitového řetězce. Automaticky lze odvozovat bitové řetězce až do délky nejdelšího explicitně uvedeného bitového řetězce. Pokud uživatel v definici abecedy uvedl pouze symboly s implicitním odvozením bitového řetězce, je nutné z mohutnosti abecedy určit délku bitového řetězce.

Každý bit bitového řetězce lze v paměti reprezentovat pomocí dvojice bitů. Bitový řetězec je reprezentován statickým polem datového typu `char`, přičemž každý prvek pole uchovává právě čtyři bity bitového řetězce. Délka pole je omezena konstantou `BITSTR_SIZE`. Efektivní délka pole je o jeden znak menší – je používán znak `'\0'` pro detekci konce bitového řetězce podobně jako u textových řetězců v jazyce C.

Jak si můžete všimnout, soubor nesoucí definici abecedy má velice jednoduchou strukturu, což byl hlavní důvod pro zakázání vkládání mezer v rámci definice jednoho symbolu. Mezery (v libovolném nenulovém počtu) slouží pouze jako oddělovač jednotlivých symbolů v souboru s definicí abecedy.

Příklad 5.1 Definice abecedy

Obsah souboru s definicí abecedy:

$$a \qquad b \qquad c(0101) \qquad d(101)$$

Načtení abecedy z tohoto souboru by proběhlo zcela korektně. Symbol c má zadán nejdelší bitový řetězec a vzhledem k malému počtu symbolů zůstane nezměněn. Situace je odlišná pro symbol d, který bude v rámci kontroly překryvu symbolů doplněn na tvar $d(101X)$. Pro symboly a a b bude automaticky odvozen unikátní bitový řetězec v závislosti na implementovaném algoritmu.
Jiná situace by nastala ve chvíli, když by byl do výše zmíněného souboru přidána následující definice symbolu:

$$e(010X)$$

Symbol e by neprošel kontrolou překryvu symbolů, protože bitové řetězce 0101 a 010X považujeme za shodné (bitový řetězec 010X totiž zastupuje řetězce 0100 a 0101).

V rámci načítání abecedy bude provedena *kontrola překryvu symbolů*. Pokud by byly nalezeny dva znaky, jejichž identifikátory nebo bitové řetězce lze zaměnit, bude systémem nahlášena chyba. V rámci kontroly překryvu symbolů jsou bitové řetězce doplněny zástupným znakem X na délku nejdelšího uvedeného bitové řetězce. Teprve po kontrole překryvu symbolů jsou automaticky odvozovány bitové řetězce pro symboly, u kterých nebyly bitové řetězce explicitně specifikovány.

Se zavedením abecedy se pojí jedna komplikace, která se týká manipulace s konečnými stromovými automaty. Bude nutné provádět další kontroly nad abecedami před provedením

některých operací, které mají na vstupu dvojici konečných stromových automatů. Například průnik dvou konečných stromových automatů vyžaduje, aby oba konečné stromové automaty byly definovány nad stejnou abecedou, neboť GTA knihovna nástroje MONA implicitně předpokládá, že všechny instance konečných stromových automatů, se kterými se pracuje, jsou definovány nad shodnou abecedou.

5.1.2 Definice konečného stromového automatu

Definice konečného stromového automatu bude oproti původnímu formátu používanému pro definování konečných stromových automatů mnohem jednodušší. Automat bude, podobně jako abeceda, definován v samostatném souboru, jehož tvar je znázorněn na obrázku 5.1. Schéma uvedená na tomto obrázku zobrazuje pouze sekce, z nichž se skládá definice konečného stromového automatu. Kromě stavů definovaných v sekci States jsou implicitně definovány stavy __INIT__ a __TRAP__, přičemž první jmenovaný zde zastupuje implicitní počáteční stav a druhý jmenovaný je „sink stav“ známý z oblasti konečných automatů.

Obrázek 5.1 Schéma definice konečného stromového automatu

```
Automaton    < název automatu >
States       < seznam stavů >
Final States < seznam koncových stavů >
Transitions  < seznam přechodových pravidel >
```

Název automatu je libovolný identifikátor (tzn. řetězec složený z malých a velkých písmen anglické abecedy, číslic a podtržítka), který prozatím slouží pouze jako základní popis automatu. *Seznam stavů* se skládá z jedinečných identifikátorů stavů konečného stromového automatu, které jsou navzájem oddělené bílými znaky. *Seznam koncových stavů* je tvořen identifikátory předem definovanými v sekci States, které jsou odděleny bílými znaky. *Seznam přechodových pravidel* tvoří přechodová pravidla oddělená bílými znaky, přičemž pravidla mohou nabývat tvaru $a->q$ nebo $f(q_1, q_2)->q$, kde $a, f \in \Sigma$ a q, q_1, q_2 jsou stavy daného konečného stromového automatu. Pravidla tvaru $a->q$ jsou chápána jako $a(\text{__INIT__}, \text{__INIT__})->q$. Při použití této definice je nutné zaručit, aby byl zadaný automat deterministický a úplný. V případě, že zadaný automat není deterministický, je nahlášena chyba. Naopak, pokud není zadaný automat úplný, GTA knihovna provede doplnění, která je založená na doplnění pravidel vedoucích do implicitního stavu – v našem případě to vždy bude stav __TRAP__. Na implicitně definovaný stav __TRAP__ je kladeno jedno omezení – nemůže být uveden v sekci Final States, tzn. nelze jej vložit do množiny koncových stavů.

Pro všechny konečné stromové automaty bude použit triviální guide, který se automaticky vygeneruje. Zadaný konečný stromový automat bude vždy umístěn do universa <univ>.

5.1.3 Definice stromového převodníku

Konkrétně se jedná o definice stromových převodníků zachovávajících strukturu, které pracují zdola nahoru. Definice stromového převodníku je uchovávána v samostatném textovém souboru, jehož struktura (znázorněna na obrázku 5.1) vychází z definic konečných stromových automatů.

Obrázek 5.2 Schéma definice stromového převodníku

```
Transducer     < název převodníku >
States         < seznam stavů >
Final States   < seznam koncových stavů >
Transductions  < seznam převodních pravidel >
```

Když pomineme odlišná klíčová slova, liší se soubor s definicí stromových převodníků pouze tím, že namísto přechodových pravidel jsou zadávána převodní pravidla, která mohou nabývat tvaru $a/b->q$ nebo $a/b(q_1, q_2)->q$, kde a je symbol ze vstupní abecedy, b je symbol z výstupní abecedy převodníku a q, q_1, q_2 jsou stavy daného stromového převodníku.

5.2 Návrh datových struktur

Knihovna, která bude pro GTA knihovnu nástroje MONA zpracovávat nově navrhovaný vstupní a výstupní formát, bude nutně definovat celou řadu datových struktur. Přehled navržených datových struktur s krátkým popisem:

- tSymbol reprezentuje jeden symbol abecedy. Tato struktura ponese informace o identifikátoru a bitovém řetězci daného symbolu. Nad touto strukturou budou implementovány funkce pro porovnávání dvojice symbolů podle identifikátoru a podle bitového řetězce.
- tAlphabet zastupuje množinu symbolů (implementovanou jako lineární jednosměrně vázaný seznam). Na strukturu reprezentující abecedu nejsou kladeny prakticky žádné nároky na datové složky, nepočítáme-li odkazy na symboly. Mezi funkcemi pro práci s abecedou budou nutně definovány funkce kontrolující unikátnost identifikátorů a bitových řetězců všech symbolů v rámci abecedy.
- tState slouží pro reprezentaci stavu konečného stromového automatu. Nejdůležitějšími složkami této struktury jsou identifikátor (pojmenování) stavu a informace o tom, zdali spadá daný stav do množiny konečných stavů.
- tStateSet odpovídá množině stavů konečného stromového automatu. Je nutné kontrolovat to, že v množině stavů nebudou uloženy dva stavy se stejným identifikátorem.
- tRule reprezentuje jedno z přechodových, resp. převodních pravidel. Tato datová struktura tedy ponese odkaz na jeden symbol, resp. dva symboly, a odkazy na trojici stavů (stavy z obou podtermů a stav ohodnocující danou pozici v termu).
- tRuleSet uchovává jednotlivá pravidla. Tato struktura bude v paměti uchovávána pouze do doby, než bude vytvořen odpovídající GTA.
- tAutomaton ponese, ať už přímo nebo nepřímo, odkazy všechny výše uvedené datové struktury. Vyjímkou jsou pouze struktury tRuleSet a tRule, které budou nahraženy odkazem na strukturu GTA, která bude reprezentovat přechodovou funkci daného konečného stromového automatu.

- tTerm reprezentuje term, který je tvořen pouze nulárními a binárními symboly. Term lze předložit na vstup konečného stromového automatu pro získání běhu, který může být přijímající (pokud term patří do jazyka přijímaného konečným stromovým automatem) nebo nepřijímající.
- tTransducer reprezentuje stromový převodník, který je oproti struktuře tAutomaton rozšířen o dvojici abeced.

5.3 Návrh interaktivního rozhraní

Jedním z požadavků je také implementace rozhraní, které by umožňovalo manipulovat s konečnými stromovými automaty interaktivně, tzn. bez nutnosti kompilace a spuštění programu.

Pro implementaci interpretu byl vybrán programovací jazyk *Haskell* [16], což je moderní a čistě funkcionální jazyk založený na redukční strategii označované jako *lazy evaluation**. Mezi nejrozšířenější implementace jazyka Haskell patří Glasgow Haskell Compiler (GHC) [17]. Součástí distribuce GHC je také inkrementální kompilátor GHCi (někdy též označovaný jako interpret), který poskytuje interaktivní rozhraní ve formě příkazové řádky. Platforma GHC je obecně považována za nejvíce rozšířenou a nejstabilnější implementaci programovacího jazyka Haskell, která je v současné době dostupná. Důkazem tohoto tvrzení je skutečnost, že *Haskell Platform*† [18] je postaven na kompilačním systému GHC. Mé řešení interaktivního rozhraní je postavené na výše zmiňovaném inkrementálním kompilátoru GHCi z kompilačního systému GHC.

Jazyk Haskell podporuje standard *Foreign Function Interface* (FFI), díky kterému lze z Haskellu volat funkce napsané v jiném programovacím (typicky v jazyce C) a naopak. Díky podpoře FFI lze implementovat knihovnu v jazyce C, zatímco v Haskellu bude vytvořena pouze vrstvu definující uživatelské rozhraní.

Při použití GHCi ve spojení s jinými programovacími jazyky pomocí FFI je potřeba vědět, že lze importovat binární soubory pouze ve formě objektů nebo dynamických knihoven. GHCi tedy neumožňuje importovat statické knihovny. Manipulace s několika desítkami zkompilovaných objektů rozmístěných v různých adresářích je relativně náročná na správu, a tak byla zvolena kompilace do dynamické knihovny, která bude zahrnovat moduly již zmiňovaných knihoven nástroje MONA společně s moduly implementujícími vlastní knihovnu pracující s novými formáty.

GHCi byl vybrán mj. z toho důvodu, že implementuje rozšíření umožňující použití klauzule let v interaktivním režimu. Jedná se o rozšíření, které není definováno v rámci standardů jazyka Haskell, díky kterému lze na globální úrovni pojmenovávat výrazy jazyka Haskell. Pouze díky tomuto rozšíření je možné v příkazové řádce, kterou poskytuje GHCi, pracovat s ukazateli na datové struktury spravované výhradně funkcemi implementovanými v jazyce C. Použití jazyka Haskell při vývoji se dále věnuje podkapitola 6.2.

*Tento pojem zde nebude překládán.

†*Haskell Platform* je kolekce softwaru, nástrojů a knihoven, které společně utvářejí základní platformu pro používání a vývoj aplikací pomocí programovacího jazyka Haskell. Základní filozofií Haskell platform by šlo vyjádřit pomocí motta: „Mít po ruce vše potřebné.“

5.4 Požadavky na implementaci a poskytované operace

Mezi základními požadavky na implementaci knihovny pro manipulaci s novými formáty vstupu a výstupu byl požadavek na použití programovacího jazyku C.

Důležitým požadavkem na knihovnu a její interaktivní rozhraní je umožnit přímou práci s deterministickým konečnými stromovými automaty pracujícími zdola nahoru, jejichž přechodová funkce je implementována pomocí knihoven nástroje MONA. Následuje výčet standardních operací, které je nutné implementovat v rámci této knihovny:

- vytvoření konečného stromového automatu, který vznikne sjednocením dvojice zadaných konečných stromových automatů;
- vytvoření konečného stromového automatu, který vznikne průnikem dvojice zadaných konečných stromových automatů;
- vytvoření konečného stromového automatu, který vznikne komplementem zadaného konečného stromového sutomatu;
- vytvoření konečného stromového automatu, který vznikne aplikací zadaného stromového převodníku zachovávajícího strukturu na zadaný konečný stromový automat.

Tento výčet samozřejmě nezahrnuje operace jako import a export konečných stromových automatů a stromových převodníků zachovávajících strukturu z/do souborů navrhovaného formátu, tj. soubory založené na výčtu pravidel.

Kapitola 6

Realizace a testování

Tato kapitola popisuje detailní návrh a realizaci prototypu aplikace, která se skládá ze dvou hlavní částí. První částí aplikace je knihovna *monalib*, pro jejíž implementaci byl použit programovací jazyk C (podkapitola 6.1). Následuje popis vlastní implementace interaktivního rozhraní pomocí programovacího jazyka Haskell (podkapitola 6.2). Závěr kapitoly se zabývá přípravou testů a testováním implementace knihovny *monalib* (podkapitola 6.3).

6.1 Realizace knihovny *monalib*

Knihovnu *monalib* je nutné kompilovat jako dynamickou knihovnu, aby ji bylo možné využívat z inkrementálního kompilátoru GHCi. Knihovna je sestavována z celé řady modulů, přičemž část těchto modulů byla implementována v rámci této diplomové práce (jejich popis je uveden v sekci 6.1.1) a zbytek knihovny *monalib* je tvořen moduly z knihoven nástroje MONA, které byly popsány v podkapitole 3.3.

6.1.1 Rozdělení na moduly

Pro knihovnu *monalib* byla implementována celá řada modulů, jejichž stručný popis je uveden v následujících odstavcích.

error
: Tento modul obsahuje implementaci funkcí, které slouží pro ladící výpisy (printDebug), varovné výpisy (printWarning) a výpisy chyb (printError). V hlavičkovém souboru k tomuto modulu lze pomocí několika definic preprocesoru upravit chování těchto funkcí (např. deskriptor souboru pro daný typ výpisu, tisk prefixu před výpisy nebo povolení/zákaz ladících výpisů).

alphabet, symbol
: Tato dvojice modulů reprezentuje ohodnocenou abecedu a její prvky, tj. symboly. V nástroji MONA jsou symboly reprezentovány pouze jako posloupnost binárních čísel, která popisuje cestu skrze BDD, kdežto symbol definovaný v modulu symbol přiřazuje této posloupnosti (označované jako bitový řetězec) unikátní identifikátor, díky čemuž přináší uživateli řádově mnohem lepší čitelnost. V rámci modulu symbol jsou definovány především operace pro práci s bitovými řetězci a porovnávání různých složek symbolů. Modul alphabet pak reprezentuje abecedu jako lineární jednosměrně vázaný seznam. Nad abcededou je definována celá řada operací: test na členství symbolu

v rámci dané abecedy, generování unikátních bitových řetězců, testování překryvu symbolů atd.

fta
: V rámci tohoto modulu jsou definovány různé datové struktury, které jsou následně využívány pro realizaci konečných stromových automatů a stromových převodníků, s výjimkou abecedy (ta je definována v samostatném modulu alphabet). Modul fta obsahuje definice datových struktur tState, tStateSet, tRule, tRuleSet a základní funkce pro práci s nimi.

lex, alphabet-lex, fta-lex
: Jedná se o trojici modulů starajících se o načítání vstupních souborů podle formátu, který je navrhnut v podkapitole 5.1. Modul lex obsahuje definici tokenu, který je využíván ve zbývajících dvou modulech. Modul alphabet-lex implementuje parser pro načítání abecedy ze souborů a modul fta-lex implementuje parsery, které ze souborů načítají konečné stromové automaty a stromové převodníky.

term
: Modul term obsahuje definice termu nad ohodnocenou abecedou. Term je načítán z textového řetězce, který je generován gramatikou s pravidly:

$$\begin{aligned} \mathtt{Term} &\rightarrow \mathtt{id} \\ \mathtt{Term} &\rightarrow \mathtt{id}(\mathtt{Term}, \mathtt{Term}) \end{aligned}$$

  kde id je terminál zastupující identifikátor symbolu a počátečním symbolem je nonterminál Term.

lib
: Hlavním modulem je lib, který spojuje funkčnost všech výše zmiňovaných modulů. Zde je definován konečný stromový automat (tAutomaton) a stromový převodník (tTransducer). V rámci tohoto modulu je implementována veškerá spolupráce s knihovnami nástroje MONA. Především pak operace nad konečnými stromový automaty.

Všechny výše uvedené moduly jsou opatřeny dokumentačními komentáři ve formátu Javadoc, z nichž je generována programová dokumentace pomocí nástroje *Doxygen* [19]. Tato dokumentace je, včetně diagramů závislosti mezi moduly, distribuována společně s vlastní knihovnou *monalib*.

6.1.2 Konstrukce konečného stromového automatu

Konečný stromový automat je v knihovně *monalib* reprezentován datovou strukturou tAutomaton. Z pohledu implementace jsou jejími nejpodstatnějšími složkami vstupní abeceda, množina stavů a symbolická reprezentace přechodové funkce. Hlavním cílem bylo explicitní přiřazení unikátních identifikátorů, které případným uživatelům nástroje MONA zjednoduší práci se symbolickou reprezentací přechodové funkce. Z pohledu uživatele je rozhodně přívětivější popsat symbol vstupní abecedy identifikátorem složených ze dvou či třech znaků než relativně dlouhou posloupností znaků 0, 1 a X, přičemž překryv těchto posloupností může výrazně ztěžovat orientaci v popisu konečných stromových automatů. Stavy automatu jsou v symbolické reprezentaci použité GTA knihovnou nástroje MONA označeny nezápornými celými čisly. Také stavům explicitně přiřazuji identifikátor, přičemž symbolické označení stavu je chápáno jako index do pole datového typu tState.

Nejprve jsem pro konstrukci přechodové funkce využíval postup, který je uveden v algoritmu 1 (viz sekce 3.3.2). Při použití tohoto postupu je při posledním kroku konstrukce přechodové funkce volána funkce gtaReachable (z GTA knihovny), která odstraní nedostupné stavy a seřadí dostupné stavy. Díky těmto optimalizacím nebylo možné zachovat explicitní označení stavů při exportování definice konečného stromového automatu.

Řešením tohoto problému byla konstrukce symbolické přechodové funkce na nižší úrovni, tzn. využití funkcí definovaných v BDD knihovně nástroje MONA pro konstrukci sdíleného MTBDD. Algoritmus konstrukce sdíleného MTBDD je zachycen v algoritmech 2, 3 a 4. Podotkněme, že algoritmy prezentované v následujících odstavcích jsou uváděny ve značně abstraktní formě. Kvůli celkové představě o algoritmech byly v uváděných pseudokódech algoritmů ponechány alespoň náznaky dynamické alokace paměti.

Algoritmus 2 zachycuje první kroky konstrukce sdíleného MTBDD, během kterých jsou nejprve provedeny potřebné alokace paměti (řádky 1–3). Poté je provedena inicializace pole `final` (řádky 4–7), do něhož jsou poznačeny koncové stavy – koncové stavy jsou označeny hodnotou 1, kdežto nekoncové stavy jsou označeny hodnotou -1. Nakonec je pro každý stavový prostor zavolána funkce Construct-Statespace (řádky 8–9). V rámci algoritmu je zanedbána inicializace jednotlivých stavových prostorů (např. alokace matice `behaviour`, která nese perzistentní odkazy na kořeny sdíleného MTBDD).

Algoritmus 3 popisuje konstrukci sdíleného MTBDD pro stavový prostor <univ>. Tento prostor je nejvíce reprezentativní, neboť pouze v tomto stavovém prostoru je generován netriviální sdílený MTBDD. V ostatních stavových prostorech (<hat> a <dummy>) jsou vytvářeny triviální sdílené MTBDD, které jsou tvořeny jen a pouze listovými BDD uzly. Také v tomto algoritmu se nejprve setkáváme s alokací paměti (řádky 1–4). Hlavní částí algoritmu je cyklus přes všechny možné dvojice stavů. V tomto cyklu jsou pro každou dvojici stavů aplikovány následující kroky:

- Vytvoření pracovního sdíleného MTBDD `bddm` (řádky 6–7), do kterého budou následně ukládány veškeré cesty vedoucí z dané dvojice stavů $(\mathtt{l}, \mathtt{p})$.
- Následně jsou v `bddm` vytvářeny listové BDD uzly pro všechny stavy z množiny Q_d. Přitom jsou do lokálního dynamicky alokovaného pole `states` ukládány perzistentní odkazy, tj. hodnoty datového typu bdd_handle, na nové vytvářené listové BDD uzly (řádky 8–9).
- Poté jsou pro všechna přechodová pravidla s počátečními stavy $(\mathtt{l}, \mathtt{p})$ vytvořeny cesty v rámci `bddm` (řádky 10–15). Cesty v `bddm` jsou vytvářeny pomocí funkce Make-Path a kořeny (vstupní body) těchto cest jsou ukládány do globálního pole `paths`. Zároveň je poznamenáván počet nalezených pravidel (čítač byl samozřejmě nejprve vynulován).
- Po průchodu všech pravidel je potřeba redukovat `bddm` tak, aby neobsahovalo duplicitní BDD uzly. Po konstrukci více než jedné cesty pomocí funkce Make-Path obsahuje `bddm` celou řadu duplicitních BDD uzlů, a tak je potřeba provést redukci `bddm`. Kroky redukce jsou v závislosti na počtu nově vytvořených cest následující:
  - Nebylo-li pro počáteční stavy $(\mathtt{l}, \mathtt{p})$ přidáno žádné pravidlo (řádek 17), pak není potřeba provádět žádnou redukci. Za těchto okolností pouze nastavíme ukazatel `ptr` na listový BDD uzel reprezentující stav __TRAP__, jenž je v algoritmech označován $\mathtt{q}_\emptyset$. Za povšimnutí stojí použití makra BDD_ROOT, které pro daný

Algoritmus 2 Pseudokód konstrukce přechodové funkce pomocí BDD knihovny

```
CONSTRUCT-DELTA(fta)
 1  fta.gta ← new GTA
 2  fta.gta.final ← new array [|Q|]
 3  fta.gta.ss ← new array [|D|]
 4  for ∀q∈Q
 5      do if q∈Q_f
 6            then fta.gta.final[q] ← 1
 7            else fta.gta.final[q] ← (−1)
 8  for ∀d∈D
 9      do CONSTRUCT-STATESPACE(fta, d)
```

Algoritmus 3 Psudokód konstrukce přechodové funkce stavového prostoru <univ>

```
CONSTRUCT-STATESPACE(fta, d = <univ>)
 1  states ← new array [|Q|]
 2  fta.gta.ss[d].behaviour ← new array [|Q_d × Q_d|]
 3  fta.gta.ss[d].bddm = bdd_new_manager(počáteční velikost hashovací tabulky,
 4                                       rezerva)
 5  for ∀(l,p)∈Q_d × Q_d
 6      do bddm ← bdd_new_manager(počáteční velikost hashovací tabulky,
 7                                rezerva)
 8         for ∀q∈Q_d
 9             do states[q] ← bdd_handle_find_leaf_hashed_add_root(bddm, q)
10         num ← 0
11         for ∀a∈Σ
12             do if δ_d(l,p)(a) ≠ nedef.
13                   then paths[num] ← MAKE-PATH(bddm, states[δ_d(l,p)(a)],
14                                               states[q_∅], λ(a), |λ(a)|)
15                        num ← num + 1
16         if num = 0
17           then ptr ← BDD_ROOT(bddm, states[q_∅])
18           else ptr ← BDD_ROOT(bddm, paths[0])
19         if num > 1
20           then bdd_make_cache(bddm, velikost cache, rezerva)
21                for i ← 1 to num
22                    do bdd_apply2_hashed(bddm, ptr, bddm,
23                                         BDD_ROOT(bddm, paths[i]),
24                                         bddm, &fn_unite)
25                bdd_kill_cache(bddm)
26         bdd_prepare_apply1(bddm)
27         bdd_apply1(bddm, ptr, fta.gta.ss[d].bddm, &fn_identity)
28         BEH(fta.gta.ss[d], l, r) ← BDD_LAST_HANDLE(fta.gta.ss[d].bddm)
29         bdd_kill_manager(bddm)
30  free states
```

perzistentní odkaz (tedy hodnotu datového typu bdd_handle) navrací odpovídající volatilní odkaz (tj. hodnota datového typu bdd_ptr). Použití tohoto makra je ve zdrojovém kódu poměrně častě, což je způsobené především tím, že většina funkcí a maker pro manipulaci se sdílenými MTBDD, které poskytuje BDD knihovna nástroje MONA, vyžadují parametry datového typu bdd_ptr.

- Po přidání jediného pravidla pro počáteční stavy $(\mathtt{l}, \mathtt{p})$ také nebude prováděna redukce, avšak nastaví se ukazatel ptr na kořen jediné cesty, kterou bddm obsahuje (řádek 18).
- Teprve v tom případě, když jsou pro počáteční stavy $(\mathtt{l}, \mathtt{p})$ přidána dvě a více pravidel, bude prováděna redukce (řádky 18 a 20–25). Redukce sdíleného MTBDD je realizována aplikací binární funkce fn_unite na dvě cesty z různých kořenů sdíleného MTBDD. Před zavoláním funkce bdd_apply2_hashed je potřeba připravit vyrovnávací paměť (řádek 20) a po provedení všech potřebných volání funkce bdd_apply2_hashed je vhodné, aby byla tato vyrovnávací paměť korektně uvolněna (řádek 25).

  Obrázek 6.1 ilustruje příklad redukce sdíleného MTBDD. V části (a) je zobrazen stav sdíleného MTBDD před provedením redukce, kdežto v části (b) je uveden výsledek získaný redukcí. Na obrázku je uvedena přechodová funkce konečného stromového automatu, který má trojici stavů. Listový BDD uzel 0 reprezentuje počáteční stav (tj. __INIT__); 1 reprezentuje stav __TRAP__; 2 reprezentuje stav q. Stav uvedený v části (a) zachycuje bddm po vytvoření trojice cest (pomocí volání funkce MAKE-PATH) pro počáteční stav $(0, 0)$, přičemž kořeny jednotlivých cest jsou od sebe navzájem odlišeny pomocí apostrofů. Jedná se o cesty popsané bitovými řetězci 11, 01 a 00, které vedou do stavu q. Jak si můžete všimnout, všechny nedefinované cesty jsou svedeny do stavu __TRAP__. Všimněme si, že po provedení redukce, která bude dokončena po dvou volání funkce bdd_apply2_hashed, se sníží počet nelistových BDD uzlů z původních pěti na pouhé dva BDD uzly. Avšak hlavním důvodem, proč je nutné provádět redukci je ten, že pro každou dvojici stavů potřebujeme právě jeden kořen (vstupní bod do sdíleného MTBDD).

- Po provedení redukce bddm je potřeba sjednotit jej se sdíleným MTBDD reprezentujícím přechodovou funkci pro daný stavový prostor, tj. fta.gta.ss[d].bddm (řádky 26 až 27). Sjednocení dvojice sdílených MTBDD se provádí aplikací unární funkce fn_identity, které je v BDD knihovně realizováno voláním funkce bdd_apply1. Před samotným zavoláním funkce bdd_apply1 je nutné provést přípravu aplikce, což obstará funkce bdd_prepare_apply1.

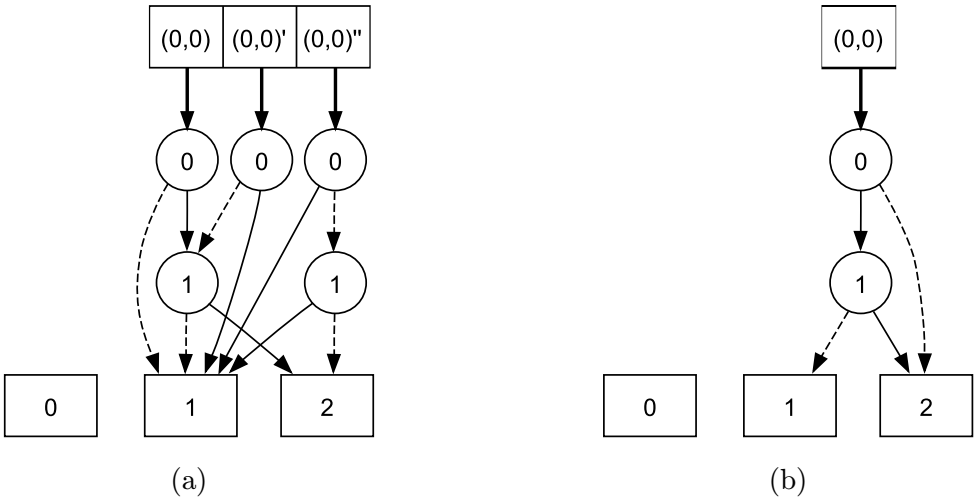
- Nyní jsme ve stavu kdy jsou v hlavním sdíleném MTBDD doplněny všechny uživatelem definované cesty, avšak je potřeba uložit do matice behaviour perzistentní odkaz na kořen posledních vkládaných cest (řáden 28). Tato akce odpovídá použití maker BEH a BDD_LAST_HANDLE. Makro BEH vrací odkaz do matice behaviour. Pomocí makra BDD_LAST_HANDLE je zpřístupněn poslední perzistentní odkaz vkládaný do pole bdd_roots.

- Po provedení jedné iterace cyklu je potřeba uvolnit bddm (řádek 29).

Posledním krokem v rámci tohoto algoritmu je uvolnění dynamicky alokovaného pole states (řádek 30), které neslo prezistentní odkazy na listové BDD uzly v rámci bddm. Při reali-

zaci zde popisovaného algoritmu je využito lokální staticky alokované pole, do něhož jsou ukládány ukazatele na pravidla před vytvořením cest v `bddm`.

Obrázek 6.1 Zachycení průběhu konstrukce sdíleného MTBDD



Posledním algoritmem, kterému je v rámci této sekce věnován prostor, je algoritmus 4, pomocí kterého jsou v `bddm` konstruovány cesty. Pro konstrukci cesty byl zvolen postup zdola nahoru, kdy se začíná od listového BDD uzlu, který reprezentuje cíl dané cesty, a pokračuje se směrem k počátečnímu uzlu. Výhodou tohoto přístupu je ta skutečnost, že není nutné uchovávat v poli bdd_roots větší počet perzistentních odkazů. Tento algoritmus vychází z předpokladu, že v `bddm` jsou již uloženy všechny listové BDD uzly. Nejprve se nastaví odkaz `ptr` cílový listový BDD uzel (řádek 1). Dále je cyklus, který prochází bitový řetězec ve směru od konce (řádky 2–11). Bity s hodnotou X jsou ignorovány, avšak pro hodnoty 0 a 1 je volána funkce bdd_find_node_hashed_add_root, jejíž chování je následující:

- Pokud v `bddm` najde BDD uzel, který odpovídá požadavkům, tzn. porovnává bit na pozici `i` a nese odkazy na dané uzly (viz 2. a 3. argument této funkce), vrátí volatilní odkaz na nalezený BDD uzel. Zároveň vloží perzistentní odkaz na tento uzel do pole bdd_roots.
- V opačném případě do hashovací tabulky v rámci `bddm` vloží nový BDD uzel, který odpovídá požadavkům. A poté vloží perzistentní odkaz na tento uzel do pole bdd_roots.

Nyní drží `ptr` volatilní odkaz na hledaný, resp. vložený, BDD uzel. V hlavičkovém souboru lib.h je definováno makro BDD_DROP_ROOT, pomocí něhož se z pole bdd_roots odstraní perzistentní odkaz, který do něj byl vložen jako poslední (řádky 7 a 11). Po průchodu bitovým řetězcem nese `ptr` volatilní odkaz na kořen cesty, která byla nově vytvořena. Voláním makra BDD_ADD_ROOT (řádek 14) uložíme do pole bdd_roots perzistentní odkaz na kořen nově vytvořené cesty, který je vrácen jako výsledek volající funkci (řádek 15).

V části (a) obrázku 6.1 je znázorněn stav `bddm` po vytvoření trojice cest pomocí funkce popisované algoritmem 4.

Implementace funkcí realizujících textový výstup, tj. tisk definic konečných stromových automatů na standardní výstup nebo export definic do souborů, využívají funkce

Algoritmus 4 Psudokód konstrukce cesty uvnitř BDD bddm

```
MAKE-PATH(bddm, leaf, def, bitstring, length)
 1  ptr ← BDD_ROOT(bddm, leaf)
 2  for i ← length − 1 downto 0
 3      do switch
 4           case bitstring[i] = '0':
 5                ptr ← bdd_find_node_hashed_add_root(bdd
 6                                          ptr, BDD_ROOT(bddm, def), i)
 7                BDD_DROP_ROOT(bddm)
 8           case bitstring[i] = '1':
 9                ptr ← bdd_find_node_hashed_add_root(bdd
10                                          BDD_ROOT(bddm, def), ptr, i)
11                BDD_DROP_ROOT(bddm)
12           case bitstring[i] = 'X':
13                /* ignorování zástupných znaků */
14  BDD_ADD_ROOT(bddm, ptr)
15  return BDD_LAST_HANDLE(bddm)
```

bdd_make_paths z BDD knihovny nástroje MONA. Tato funkce očekává dvojici parametrů: bdd_manager a kořen, ve kterém mají cesty začínat. Jako výsledek je vráceno pole struktur reprezentujících cesty sdíleným MTBDD, které začínají v zadaném kořenu. Po zpracování lze takto získané pole struktur uvolnit pomocí funkce kill_paths. Tyto funkce jsou navrženy tak, aby nebyla tisknuta ta pravidla, která vedou do stavu __TRAP__. Díky tomu je výsledný popis automatu kratší a přehlednější, aniž by byla došlo ke ztrátě informace.

6.1.3 Operace nad konečnými stromovými automaty

Nejprve se podíváme na operace *sjednocení* a *průniku* stromových jazyků, což jsou binární operace nad konečnými stromovými automaty. Je nutné, aby byly operandy těchto operací definovány nad vstupními abecedami, které nejsou v kolizi. Uvažujme, že dvě abecedy jsou v kolizi právě tehdy, když v jedné abecedě existuje alespoň jeden symbol takový, že se od symbolu z druhé abecedy liší pouze v identifikátoru nebo pouze v bitovém řetězci. Prakticky se abecedy v kolizi projeví tak, že v abecedě, kterou získáme sjednocením těchto dvou abeced, bude detekován překryv, přestože v původních abecedách překryv detekován nebyl. Aplikací operace sjednocení, resp. průniku, na dvojici konečných stromových automatů získáme nový konečný stromový automat, který odpovídá následujícímu předpisu:

- Abeceda vznikne sjednocením abeced, nad kterými jsou definovány operandy. Odtud plyne požadavek, aby abecedy operandů nebyly navzájem v kolizi.
- Konstrukci přechodové funkce obstará funkce gtaProduct, která očekává tři argumenty. Dva argumenty jsou ukazatele na struktury GTA vstupních operandů a posledním argumentem je požadovaná operace. Pouze posledním parametrem je tedy ovlivněno, zda se provádí sjednocení či průnik. Pro sjednocení je potřeba zadat hodnotu gtaOR a pro průnik je nutné zadat hodnotu gtaAND.
- Konstrukce množiny stavů nově vytvořeného konečného stromového automatu se prakticky skládá ze dvou částí. První část je uskutečněna nástrojem MONA při provádění předchozího bodu a druhá část je založena na explicitním přiřazení pojmenování

stavů, s nimiž pracuje přechodová funkce vytvořená nástrojem MONA. Stavům je přiřazeno označení odpovídající řetězci q_x, kde x je celé kladné číslo, které reflektuje interní pořadí stavů v rámci přechodové funkce. Výjimku tvoří stavy __INIT__ a __TRAP__, které jsou automaticky detekovány a následně je jim přiřazeno implicitní označení. Teoreticky se může stát, že nástroj MONA sjednotí stav __TRAP__ s jiným stavem. V takovém případě je však nutné exportovat všechna pravidla uvedená v symbolické reprezentaci přechodové funkce.

Nyní přistoupíme k unárním operacím *doplňku* a *minimalizace* jazyku konečného stromového automatu. Minimalizace je triviální oprace, která je postavená pouze na vytvoření nové přechodové funkce pomocí funkce gtaMinimize. Přejděme proto raději rovnou k doplňku stromového jazyka nad abecedou Σ. Doplněk stromového jazyku získáme tím, že použijeme funkci gtaNegation, avšak takto získaná přechodová funkce obsahuje mnohem větší množství přechodových pravidel, což je způsobené použitím symbolické reprezentace abecedy v GTA knihovně nástroje MONA. Symboly z abecedy jsou reprezentovány bitovým řetězcem, který je tvořen k bity. Abeceda tedy může obsahovat nejvýše 2^k symbolů, avšak většinou je počet symbolů mnohem menší. Přechodová funkce, kterou získáme použitím funkce gtaNegation, může obsahovat pravidla se symboly, které nepatří do dané abecedy. Proto je nutné udělat průnik přechodové funkce, kterou jsme získali použitím funkce gtaNegation, s přechodovou funkcí konečného stromového automatu, který reprezentuje stromový jazyk Σ^*. Přičemž stromový jazyk Σ^* obsahuje všechny termy nad danou abecedou Σ. Po aplikaci těchto dvou kroků získáme konečný stromový automat reprezentující stromový jazyk, který vznikl doplňkem stromového jazyku reprezentovaného původním operandem.

Dále je nad konečnými stromovými automaty implementována dvojice testů. Prvním je *test prázdnosti stromového jazyka*. Tento test očekává jako operand konečný stromový automat a vyhodnocení je založeno na použití funkce gtaMakeExample, která je poskytována nástrojem MONA pro generování příkladů termů, jenž jsou přijímány zadaným konečným stromovým automatem. Pokud zadaný konečný stromový automat nepřijímá žádné termy, pak je stromový jazyk reprezentovaný tímto automatem prázdný a funkce gtaMakeExample vrací NULL (v opačném případě vrací ukazatel na struktru). Druhý implementovaný test je založen na následujícím vztahu:

$$L(M_1) \subseteq L(M_2) \iff L(M_1) \cap \overline{L(M_2)} = \emptyset$$

Tento vztah lze interpretovat takto: uvažujeme dva stromové jazyky $L(M_1)$, $L(M_2)$ reprezentované konečnými stromovými automaty M_1, M_2, pro které platí, že jazyk $L(M_1)$ je podmnožinou jazyka $L(M_2)$ právě tehdy, když jsou jazyk $L(M_1)$ a doplněk jazyka $L(M_2)$ navzájem disjunktní. Průnik, doplněk a test prázdnosti jsou operace, které již máme implementovány, a tak můžeme realizovat test vztahu:

$$L(M_1) \subseteq L(M_2)$$

který očekává dvojici operandů, konečné stromové automaty M_1 a M_2.

Poslední implementová operace, která pracuje nad konečnými stromovými automaty, slouží pro *testování členství zadaného termu*. Term je zadáván v textové formě definované gramatikou, která je uvedena v rámci sekce 6.1.1. Běh nad zadaným termem lze rovněž vypsat ve formě termu ohodnoceného stavy konečného stromového automatu.

6.1.4 Použití globálních proměnných a sestavení dynamické knihovny

Ve zdrojovém souboru lib.c je definováno několik globálních proměnných. Globální proměnná guide je zprvu neinicializovaná a její inicializace na triviální guide je provedena ve chvíli, když je poprvé zavolána funkce aux_newFTA z modulu lib. Informace o tom, zda je již triviální guide vytvořen, je uchovávána v další globální proměnné guide_constructed, jejíž hodnota je nenulová právě tehdy, když je proměnná guide inicializována. Poslední z využívaných globálních proměnných je my_paths, která slouží pro dočasné uložení kořenů nově vytvořených cest – odpovídá tedy proměnné `paths` v algoritmu 3. Nejvyšší možný počet nově vytvořených cest je určen hodnotou konstanty MAX_PATHS.

Při sestavení dynamické knihovny *monalib* je nutné, aby byly moduly knihovny pro správu paměti používané nástrojem MONA kompilovány s volbou -DUSE_MALLOC (viz sekce 3.3.3).

6.2 Realizace interaktivního rozhraní

Po celou dobu vývoje a testování interaktivního rozhraní byl používán kompilační systém GHC ve verzi 6.12.3 v prostředí operačních systémů MS Windows XP a GNU/Linux s jádrem verze 2.6.32. Tato verze kompilačního systému odpovídá standardu *Haskell 98* [20], což je stabilní standard, který je aktivně používán celou řadu let.

V průběhu vývoje se stala dostupná novější verze kompilačního systému GHC – verze 7.0.2, což je první implementace jazyka Haskell odpovídající standardu *Haskell 2010* [21]. Mezi nejpodstatnější změny v novějším standardu, které jsou proklamovány, patří užší integrace FFI do samotného jádra Haskellu, menší změny ve specifikaci formální gramatiky jazyka a zavedení hierarchie pojmenování modulů. Žádná z přinesených změn by prakticky neměla ovlivnit použití knihovny *monalib* s GHCi.

Inkrementální kompilátor GHCi při svém startu umožňuje načítání:

- modulů kompilovaných např. kompilátorem jazyka C (v angličtině označovaných jako *object files*),
- nebo dynamických knihoven ve formátu očekávaném daným operačním systémem (např. DLL formát používaný operačními systémy Microsoft Windows nebo SO formát používaný celou řadou operačních systémů založených na UNIXu).

Z důvodu lepší udržovatelnosti aplikace jsem se rozhodl, pro kompilaci všech používaných modulů do dynamické knihovny, která je popsána v podkapitole 6.1.

6.2.1 Rozdělení na moduly

Implementace interaktivního rozhraní se skládá z následujících modulů:

MONA.hs
: V rámci tohoto modulu jsou realizovány funkce, propojující samotné interaktivní rozhraní s knihovnou *monalib*.

MONA_const.hs
: Modul MONA_const.hs obsahuje aktuální hodnoty všech podstatných konstant rozgenerovávaných pomocí preprocesoru jazyka C při překladu modulů knihovny *monalib* (jejich přehled je uveden v sekci 6.1.1). Pro získání hodnot konstant definovaných ve zdrojových textech jazyka C je používán nástroj hsc2hs (viz sekce 6.2.4).

MONA_FFI.hs
: Jedná se o relativně malý modul, jehož jedinou funkcí je definování rozhraní pro volání funkcí jazyka C za pomocí FFI.

MONA_gen.hs
: Pomocí tohoto modulu může uživatel automaticky generovat soubory s definicemi abecedy. Uživatel pouze zadá seznam identifikátorů a cestu k nově vytvořenému souboru, do kterého bude výsledná definice abecedy uložena. Modul MONA_gen.hs exportuje pouze dvojici funkcí: generateAlphabet a generateAlphabetM. První zmiňovaná funkce generuje bitové řetězce v implicitním režimu, kdežto druhá funkce navíc očekává parametr udávající režim generátoru.

MONA_parse.hs
: Tento modul implementuje sadu funkcí provádějících lexikální analýzu nad soubory s definicemi abeced, konečných stromových automatů a stromových převodníků. Tyto funkce jsou volané před importem datových struktur z definic v souborech, aby se ověřila jejich syntax. Modul MONA_parse.hs využívá knihovny Parsec – knihovny monadických kombinátorů pro parsování.

MONA_test.hs
: Pro účely testování knihovny *monalib* byl implementován modul MONA_test.hs. Pro spuštění všech testů je potřeba zavolat funkci checkAll, která následně informuje o výsledcích všech již provedených testů. Způsob realizace testů je popsán v podkapitole 6.3.

Podobně jako zdrojové texty modulů implementovaných v jazyce C, taktéž zdrojové texty modulů implementovaných v jazyce Haskell jsou opatřeny dokumentačními komentáři, z nichž je následně generována programová dokumentace. Pro generování programové dokumentace ze zdrojových textů jazyka Haskell byl zvolen nástroj *Haddock* [22].

6.2.2 FFI rozhraní

Díky použití GHCi lze pomocí klauzule let držet v Haskellu na globální úrovni ukazatele na struktury, s nimiž pracují funkce implementované v jazyce C. Použití klauzule let pro uchování odkazů na výrazy při práci v příkazové řádce je jedním z rozšíření, která jsou poskytována kompilační systém GHC.

Vzhledem k tomu, že v rámci implementace interaktivního rozhraní není potřeba přistupovat k jednotlivým složkám používaných dynamických datových struktur, není nutné, aby Haskell znal vnitřní strukturu těchto struktur. Haskell pouze drží ukazatele na datové struktury alokované pomocí knihovny *monalib* a tyto ukazatele dále předává jako argumenty funkcím implementovaným v jazyce C.

Nicméně, Haskell poskytuje možnost, jak přistupovat ke složkám datových struktur definovaných v jiných programovacích jazycích. Pro tyto účely je v Haskellu typová třída Storable (definovaná v modulu Foreign.Storable). Postup využití typové třídy Strorable je následovný:

- Nejprve si v Haskellu nadefinujeme nový datový typ Struct, který bude reprezentovat datovou strukturu z jazyka C.
- Následně je potřeba nadefinovat Struct jako instanci typové třídy Storable, což znamená uvedení „hlavičky“:

```
instance Storable Struct where
```

za níž následují definice minimální množiny funkcí:

- Funkce sizeOf navracející velikost (v bytech) potřebného paměťového prostoru pro uchování datové struktury.
- Funkce alignment vracející hodnotu zarovnání v paměti, tj. číselnou hodnotu, kterou musí být dělitelná adresa, na níž je struktura uložena v paměti.
- Alespoň jedna z funkcí peek, peekElemOff a peekByteOff, které slouží pro čtení složek datové struktury.*
- Alespoň jedna z funkcí poke, pokeElemOff a pokeByteOff, jenž jsou určeny pro zápis složek datové struktury.*

Tento způsob přístupu ke složkám datových struktur definovaných v jiných programovacích jazycích je při praktickém použití poněkud těžkopádný. Na druhou stranu se jedná o spolehlivý způsob, který svůj účel splňuje. Zápis definic popisovaný v předchozích bodech lze částečně zjednodušit použitím nástroje *hsc2hs* (viz sekce 6.2.4).

Implementace rozhraní ke knihovně *monalib* navyžaduje definování instancí typové třídy Storable. Plně postačuje uchovávání ukazatelů z jazyka C. Pro tyto účely lze v Haskellu využít buď datový typ Ptr a nebo IntPtr (oba definované v modulu Foreign.Ptr). Z praktického hlediska jsou oba datové typy stejné, a tak jsem při realizaci využil datového typu IntPtr. Typ Ptr a je vhodnější použít pro uchování odkazů na datové typy, které jsou instancemi typové třídy Storable.

Při práci s ukazateli je nutné myslet také na to, že ukazatel je pouhá adresa do paměti. Haskell je programovací jazyk, jehož typový systém je statický a silně typovaný, což lze považovat za výhodu. Haskell zná v době překlady datové typy všech funkcí a výrazů, se kterými pracuje. Pokud chceme, aby bylo interaktivní rozhraní implementované v Haskellu typově bezpečné, vytvoříme v Haskellu nový datový typ pro každou datovou strukturu z jazyka C, s niž může být manipulováno v Haskellu. Jinak řečeno nebudeme pracovat přímo s ukazateli, ale s datovými strukturami Haskellu, uvnitř kterých budou ukazatele drženy. V implementaci interaktivního rozhraní knihovny *monalib* používám několik datových typů. Jako ukázku si zde uvedeme definici datového typu TAutomaton, který nese ukazatel na datovou strukturu tAutomaton definovanou v jazyce C:

```
type TAutomaton = IORef TAutomatonPtr

data TAutomatonPtr
   = NoAutomaton
   | Automaton PtrFTA

type PtrFTA = IntPtr
```

Na prvním řádku je pomocí klíčového slova type vytvořen alias (stejně jako pomocí operátoru typedef v jazyce C) pro datový typ IORef TAutomatonPtr. Datový typ IORef se používá pro uchování hodnot, které se mohou měnit. Zde je použita z toho důvodu, aby bylo možné změnit hodnotu ukazatele na ekvivalent hodnoty NULL po uvolnění datové struktury z paměti (voláním funkce disposeFTA). Druhá definice začíná klíčovým slovem data, které slouží

*Pokud nadefinujeme pouze některé z funkcí uvedených v tomto bodu, neznamená to, že by zbývající zůstaly nedefinované. Kompilátor jazyka Haskell je schopen automaticky odvodit definice zbývajících funkcí.

pro vytváření nových datových typů. Za klíčovým slovem data následuje pojmenování nového datového typu (v tomto případě TAutomatonPtr) a za znakem = následují jednotlivé datové konstruktury oddělené znakem |. Instanci datového typu TAutomatonPtr vytvoříme pomocí jednoho z následující dvojice datových konstruktorů:

- Datový konstruktor NoAutomaton je konkrétní hodnotou datového typu TAutomatonPtr. Tento datový konstruktor využívám jako ekvivalent k NULL v jazyce C.
- Konstruktor Automaton je parametrický konstruktor, který má jeden parametr datového typu PtrFTA. Tento fakt lze interpretovat tak, že „obalením“ ukazatele (hodnoty datového typu PtrFTA, což je alias typu IntPtr) do datového konstruktoru Automaton získáme hodnotu datového typu TAutomatonPtr.

Z tohoto rozboru použitých datových typů vyplývá ta skutečnost, že funkcím v Haskellu jsou jako argumenty převádány hodnoty datového typu TAutomaton, v rámci nichž jsou uchovávány ukazatele na datové struktury tAutomaton v jazyce C. Díky tomuto „obalení“ ukazatelů do datových typů Haskellu od sebe můžeme odlišit ukazatel na konečný stromový automat od ukazatele na stromový převodník, který je v Haskellu reprezentovaný datovým typem TTransducer. Tímto krokem je zajištěna určitá bezpečnost při manipulaci s ukazateli datových struktur knihovny *monalib*.

6.2.3 Monáda IO

Strategie vyhodnocení, kterou využívá programovací jazyk Haskell, je označována jako *lazy evaluation* a jedním z jejích nejvýraznějších rysů je schopnost zapamatovat si výsledky již vyhodnocených výrazů. Praktické úskalí této skutečnosti si ukážeme na příkladu. Představme si, že v GHCi voláme funkci newStruct a ta pomocí FFI zavolá stejnojmennou funkci implementovanou v jazyce C, která vrací novou instanci datové struktury. Pomocí let si pojmenujeme dvě instance této struktury a následně si necháme na standardní výstup vytisknout hodnoty ukazatelů získaných ze dvou nezávislých volání funkce newStruct:

```
ghci> let a = newStruct
ghci> let b = newStruct
ghci> a
123456789
ghci> b
123456789
```

Na první pohled je zřejmé, že jsme získali dva stejné ukazatele. Pojďme se společně podívat na to, co se skutečně stalo. První dva příkazy ve skutečnosti žádnou funkci nezavolají. GHCi si pouze poznamená, že když uživatel v budoucnu použije identifikátor a, tak tím myslí výraz newStruct (stejně tak b). Dalším příkazem žádáme GHCi o výpis hodnoty, na kterou se zredukuje výraz a. V tuto chvíli GHCi přes rozhraní FFI zavolá žádanou funkci v jazyce C a výsledek navrácený touto funkcí, tj. ukazatel na nově vytvořenou datovou strukturu, je předán zpět GHCi. GHCi si tuto hodnotu na globální úrovni zapamatuje jako výsledek výrazu newStruct a zároveň tuto hodnotu vytiskne na následujícím řádku. Poté chceme vypsat hodnotu výrazu b. GHCi zjistí, že výraz b je shodný s výrazem newStruct, jehož výsledek již zná. Tak tento výsledek vypíše, aniž by znovu zavolal požadovanou funkci. Takové chování nám nevyhovuje, neboť pomocí funkce newStruct nelze získat více než jednu instanci dané datové struktury.

Řešením problému z předchozího odstavce je použití monády IO, pomocí které jsou definovány veškeré vstupně výstupní operace. Pokud v Haskellu chceme např. načíst znak ze standardního výstupu, použijeme funkci getChar, jejíž *anotace* je následující:

```
getChar :: IO Char
```

Anotace v Haskellu slouží k popisu datového typu výrazu nebo funkce. Tato anotace popisuje funkci getChar, jejíž návratový typ je IO Char, přičemž pomocí dvojice dvojteček je v rámci typové anotace oddělen výraz nebo jméno funkce od inskripce, která popisuje datový typ. Datový typ Char reprezentuje jeden znak a IO značí, že funkce getChar je *monadická funkce*. Podstatné je to, že Haskell si pro monadické funkce nesmí pamatovat výsledky. Vrátíme se k našemu příkladu, avšak tentokrát uvažujme, že newStruct je monadická funkce:

```
ghci> let a = newStruct
ghci> let b = newStruct
ghci> a
123456789
ghci> b
142536159
ghci> b
159357248
```

Díky monádě IO jsme vyřešili náš problém s pamatováním mezivýsledku, ale objevil se nový problém. Při každém zavolání funkce newStruct v Haskellu dojde k vytvoření nové instance datové struktury v jazyce C. Ani takové chování nám nevyhovuje.

Řešení tohoto problému existuje a jeho princip je ve své podstatě jednoduchý. Potřebujeme funkci newStruct nadefinovat tak, aby se uvnitř chovala jako monadická funkce, avšak zvenčí by se jednalo o čistou funkci. Pro tento účel lze použít funkci unsafePerformIO. Ve zdrojovém textu modulu MONA.hs lze nalézt volání funkce unsafePerformIO uvnitř definice každé funkce, která vrací nově vytvořenou datovou strukturu (např. importFTA a union).

6.2.4 Nástroj hsc2hs

Nástroj hsc2hs lze použít pro usnadnění tvorby rozhraní s jazykem C. Nástroj hsc2hs čte zdrojový text Haskellu doplněný o speciální konstrukce a jeho výstupem je zdrojový text Haskellu, ve kterém jsou speciální konstrukce rozgenerovány na požadovaný obsah. Vstupní soubory tohoto nástroje nejčastěji nesou příponu .hsc a výstupními soubory jsou standardní zdrojové texty Haskellu, tj. jsou zakončeny příponou .hs.

Ve svém řešení využívám nástroje hsc2hs pro zpracovaní modulu MONA_const.hsc. Tento modul používá pouze trojice speciálních konstrukcí nástroje hsc2hs. Využívanými konstrukcemi jsou:

- Z jazyka C je převzata konstrukce #include, která má stejný zápis i význam.
- Za klíčovým slovem #const následuje název celočíselné konstanty definované pomocí #define ve zdrojovém souboru jazyka C, který je načten pomocí konstrukce #include. Ve výsledném souboru je tato konstrukce substituována hodnotou dané konstanty.
- Pomocí klíčového slova #const_str je uvozena konstrukce, která se chová podobně jako #const, avšak hodnota dané konstanty musí být typu řetězec.

Po použití nástroje hsc2hs získáme modul MONA_const.hs, pomocí kterého jsou zpřístupněny snad všechny prakticky použitelné konstanty definované ve zdrojových textech knihovny

monalib. V rámci tohoto modulu je definována funkce printConstFromC, která na standardní výstup vytiskne přehled podstatných konstant včetně jejich aktuálních hodnot. Pro změnu některé z konstant je nutné upravit příslušný #define, provést opětovnou kompilaci knihovny *monalib* a znovu spustit interaktivní rozhraní.

6.3 Příprava testů a testování

Veškeré testovací funkce jsou implementovány v rámci modulu MONA_test.hs. Tento modul využívá přímo rozhraní definovaného modulem MONA_FFI.hs pro testování správnosti implementace knihovny *monalib*.

Testování sestává z dvou metod testování:

- Automatické testování založené na náhodně generovaných datech, které je realizováno pomocí modulu Test.QuickCheck [23]. Tento způsob je využit pro testování některých funkcí modulu term. Jeden z testů je založen na tom, že náhodně vygenerovaný term je v Haskellu pomocí funkce show převeden do textové reprezentace. Tento řetězec je následně předán modulu term, aby z něj v jazyce pomocí funkce newTerm byla zkonstruována struktura tTerm, která je následně převedena funkcí termToString zpět do textové reprezentace. Test je založen na porovnání těchto dvou řetězců – původního vygenerovaného v Haskellu a toho nově vytvořeného v jazyce C. Při tomto testu je prověřena správnost implementace dvou funkcí implementovaných v modulu term. Pro tuto metodu je nutné nadefinovat generátoty vstupních dat, přičemž vstupní data jsou v tomto případě reprezentovanána rekurentním datovým typem Term.

- Testování založené na předem vytvořené množinách testovacích dat. Při použití této metody je test prováděn podle přesného předpisu.

První metoda demonstruje možnosti, které Haskell poskytuje v oblasti automatizovaného testování funkce. Její hlavní výhodou je využití libovolně veliké množiny náhodně vygenerovaných vstupních parametrů. Velikost množiny lze samozřejmě ovlivnit, avšak pro většinu případů je postačující provedení testu na 100 náhodně vygenerovaných vstupních parametrech testované funkce. Bohužel nelze tento způsob použít pro otestování klíčových funkcí knihovny *monalib*. Veliká komplexnost této práce nedovoluje efektivně navrhnout generátory testovacích dat, a tak je většina implementovaných testů postavena na druhé ze zde uvedených metod, tj. použití předem vytvořených vstupů pro jednotlivé testy.

Testování jsou připraveny dvě sady definic konečných stromových automatů. Základní sada se skládá z definic jednodušších konečných stromových automatů, kterých je několik desítek. Nad základní sadou testovacích vstupů jsou testovány následující vlastnosti:

- Testování termů, které patří do jazyka reprezentovaného testovaným konečným stromovým automatem. Ke každému z konečných stromových automatů je výčet přijímaných termů uchováván v samostatném textovém souboru, který na každém řádku uveden právě jeden term.

- Testování nepřijetí termů, které nepatří do jazyka, jenž je reprezentován testovaným konečným stromovým automatem. Sada těchto termů je buď generována automaticky na je využit výčet termů uvedených v textových souborech (podobně jako u předchozího bodu). Pro automatické generování termů je využíván modul Test.QuickCheck. Každý z automaticky vygenerovaných termů je neprve porovnán s přijímanými termy

(viz předchozí bod) a shodné termy jsou zahazovány. Díky „filtrování“ vygenerovaných termů jsou tyto testy výrazně časově náročnější.

- Další testované vlastnosti jsou založeny na správném provedení operací sjednocení, průniku a doplňku regulárních stromových jazyků, které jsou reprezentovány testovanými konečnými stromovými automaty. Správnost provedení těchto operací je prověřována simulací běhu nad termy, které by měli být výslednými konečnými stromovými automaty přijímány.

Pro testování je dále připravena sada definic komplexních konečných stromových automatů. Tato sada testovacích vstupů byla převzata z praktických příkladů červeno-černých stromů, se kterými se pracovalo pomocí nástroje Timbuk v rámci výzkumné skupiny VeriFIT. Vstupy z Timbuku bylo nejprve nutné upravit tak, aby odpovídali formátu, jehož návrh jsem vytvořil v rámci této práce.

Kapitola 7

Zhodnocení

Cílem této diplomové práce byl návrh a vytvoření rozhraní pro knihovny nástroje MONA, které jsou určené pro práci s konečnými stromovými automaty a stromovými převodníky. Implementace se skládála ze dvou částí, přičemž jádro rozhraní, tj. knihovna *monalib*, mělo být implementováno v programovacím jazyce C, kdežto druhá část, tj. interaktivní rozhraní ve formě příkazové řádky, mohla být implementována v libovolném programovacím jazyku. Pro implementaci interaktivního rozhraní ke knihovně *monalib* byl zvolen programovací jazyk Haskell a jako nástroj byl použit inkrementální kompilátor GHCi, který je součástí kompilačního systému GHC. V rámci modulů implementovaných v programovacím jazyku Haskell je dle FFI definováno rozhraní pro volání funkcí z knihovny *monalib*.

Práce shrnuje teoretické základy, na kterých je postaven formalismus konečných stromových automatů. Dále se zabývá existujícími nástroji umožňujícími práci s konečnými stromovými automaty, které jsou využitelné v oblasti formální verifikace. Následuje obsahově krátká kapitola, v rámci které je popisována konkrétní ukázka s definicí konečného stromového automatu pro GTA knihovnu nástroje MONA. Zbývající část práce se zabývá návrhem, realizací a testováním vytvořeného nástroje (knihovny *monalib* doplněné o interaktivní rozhraní).

Pro účely testování byl vyhražen jeden z modulů implementovaných pomocí programovacího nástroje Haskell. Část testování probíhá na základě dynamicky generovaných vstupů a zbývající testy jsou připraveny předem. Pro dynamické generování vstupů je využívána knihovna QuickCheck, jež je standardně distribuována společně s kompilačním systémem GHC.

Další vývoj implementovaného nástroje by mohl zahrnovat rozšíření funkcionality knihovny *monalib*, především pak v oblasti stromových převodníků zachovávajících strukturu. Nad knihovnami nástroje MONA lze implementovat zpětnou aplikaci stromového převodníku nebo též kompozici dvou stromových převodníků.

Při testování jsem narazil na limit, který je způsoben tím, že implementace nástroje MONA je optimalizována na 32bitovou architekturu. V průběhu konstrukce přechodové funkce konečného stromového automatu postupně dochází k fragmentaci logického adresového prostoru, a tak se od určité velikosti konečného stromového automatu nepodaří alokovat souvislý blok paměti o požadované velikosti. V tuto chvíli dochází k okamžitému ukončení, kterému předchází chybové hlášení informující o vyčerpání paměťového prostoru.

Literatura

[1] *The MONA Project* [online]. Version 1.4-13. August 5 2008 [cit. 2010-11-26]. Domovská stránka: <http://www.brics.dk/mona>.

[2] Comon, H., Dauchet, M., Gilleron, R. et al. *Tree Automata Techniques and Applications* [online]. November 18, 2008 [cit. 2010-11-26]. Dostupné na: <http://tata.gforge.inria.fr>.

[3] Rogalewicz, A., Vojnar, T. a Salagnac, P. *ARTMC – Abstract Regular Tree Model Checking* [online]. 10th November 2009 [cit. 2011-04-26]. Domovská stránka: <http://www.fit.vutbr.cz/research/groups/verifit/tools/artmc>.

[4] Habermehl, P., Holík, L., Rogalewicz, A. et al. *Forester – Tool for Verification of Programs with Pointers* [online]. [cit. 2010-11-26]. Domovská stránka: <http://www.fit.vutbr.cz/research/groups/verifit/tools/forester>.

[5] Genet, T. *Timbuk: for Reachability Analysis and Tree Automata Calculations* [online]. [cit. 2010-11-26]. Domovská stránka: <http://www.irisa.fr/celtique/genet/timbuk>.

[6] *The Caml Language* [online]. Latest update of this page: 2005-01-28 [cit. 2011-05-18]. Domovská stránka: <http://caml.inria.fr>.

[7] *GNU Library General Public License, version 2.0* [online]. Version 2. June 1991 [cit. 2011-05-18]. Text licence dostupný na: <http://www.gnu.org/licenses/old-licenses/lgpl-2.0.html>.

[8] *GNU General Public License, version 2* [online]. Version 2. June 1991 [cit. 2011-05-18]. Text licence dostupný na: <http://www.gnu.org/licenses/old-licenses/gpl-2.0.html>.

[9] Klarlund, N. a Møller, A. *MONA Version 1.4 User Manual* [online]. 2001 [cit. 2010-11-26]. Manuál dostupný na: <http://www.brics.dk/mona/mona14.pdf>.

[10] Biehl, M., Klarlund, N. a Rauhe, T. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata (WIA '96)* [online]. London: Springer Verlag, 1997 [cit. 2011-04-27]. Článek dostupný na: <http://www.brics.dk/mona/papers/algorithms-guided-tree-aut/article2.pdf>. ISBN 3-540-63174-7.

[11] Klarlund, N., Moller, A. a Schwartzbach, M. I. MONA Implementation Secrets. *International Journal of Foundations of Computer Science* [online]. 2002, roč. 13, č. 4 [cit. 2011-04-27]. S. 571–586. Článek dostupný na: <http://www.brics.dk/mona/papers/implementation-secrets/journal.pdf>. ISSN 1793-6373.

[12] Klarlund, N. a Rauhe, T. *BDD algorithms and cache misses* [online]. 1996 [cit. 2011-04-27]. Technická zpráva dostupná na: <http://www.brics.dk/mona/papers/bdd-alg-cache-miss/article.pdf>.

[13] Knuth, D. E. *The Art of Computer Programming, Svazek 4* [online]. Reading (Massachusetts): Addison-Wesley, 22 December 2008 [cit. 2011-01-08]. Jednotlivé části dostupné na: <http://cs.utsa.edu/~wagner/knuth>. ISBN 0-321-58050-8.

[14] *Eclipse Public License – v 1.0* [online]. [cit. 2011-05-19]. Text licence dostupný na: <http://www.eclipse.org/legal/epl-v10.html>.

[15] *Graphviz: Graph Visualization Software* [online]. [cit. 2011-05-09]. Domovská stránka: <http://www.graphviz.org>.

[16] *The Haskell Programming Language* [online]. last modified 17:08, 30 November 2010 [cit. 2011-05-09]. Domovská stránka: <http://haskell.org>.

[17] *GHC (The Glasgow Haskell Compiler)* [online]. [cit. 2011-05-09]. Domovská stránka: <http://haskell.org/ghc>.

[18] *The Haskell Platform* [online]. 2011.2.0.1. [cit. 2011-05-09]. Dostupná na: <http://hackage.haskell.org/platform>.

[19] Heesch, D. van. *Doxygen* [online]. v1.7.4. last modified on 30 March 2011 [cit. 2011-05-09]. Domovská stránka: <http://www.doxygen.org>.

[20] *Haskell 98 Language and Libraries: The Revised Report* [online]. December 2002 [cit. 2011-05-09]. Dostupný na: <http://www.haskell.org/onlinereport>.

[21] *Haskell 2010: Language Report* [online]. 2010 [cit. 2011-05-09]. Dostupný na: <http://www.haskell.org/onlinereport/haskell2010>.

[22] Marlow, S. *Haddock: A Haskell Documentation Tool* [online]. Sep 2 2010: Haddock version 2.8.0. [cit. 2011-05-09]. Domovská stránka: <http://www.haskell.org/haddock>.

[23] Claessen, K. a Hughes, J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* [online]. 2000, roč. 35, č. 9 [cit. 2011-05-14]. S. 268–279. Článek dostupný na: <http://portal.acm.org/citation.cfm?doid=357766.351266>. ISSN 0362-1340.

Příloha A

Manuál

Tato příloha prezentuje informace, které by měly případnému uživateli knihovny *monalib* a především pak jejího interaktivní rozhraní usnadnit orientaci. Je pravděpodobné, že manuál dodávaný společně s dokumentací zdrojových textů bude rozšiřovat obsah této přílohy.

Adresářová struktura implementace knihovny *monalib* včetně interaktivního rozhraní je znázorněna na adresářové struktuře 1.

A.1 Instalace

Nejprve je nutné ze stránek projektu MONA [1] stáhnout aktuální verzi zdrojových textů nástroje MONA. Při této akci je požadováno vyplnění krátkého formuláře, což nezabere příliš času.

Po stažení zdrojových textů je potřeba zkopírovat adresář _monalib_ do kořenového adresáře zdrojových textů nástroje MONA. Tento krok je vyžadován, protože pro sestavení dynamické knihovny *monalib* jsou potřeba zdrojové texty obsažené v adresářích BDD, GTA a Mem. Pro kompilaci zdrojových textů v těchto adresářích slouží příkaz make mona*.

Následujícím krokem je kompilace zdrojových textů knihovny *monalib* a následné sestavení dynamické knihovny *monalib*, která se spustí příkazem make*. Pokud proběhl úspěšně i tento příkaz, pak je vše připraveno, pak můžeme přistoupit k samotnému použití.

Kvůli výrazným odlišnostem operačních systémů MS Windows a implementace kompilačního systému GHC pro tyto operační systémy je potřeba provádět kompilaci a sestavení knihovny *monalib* jiným způsobem. Z toho důvodu jsou v adresáři src umístěny soubory Makefile a Makefile.win, pomocí kterých je uživatel odstíněn od rozdílů mezi operačními systémy. Soubor Makefile.win je používán pouze a jen tehdy, když je na daném operačním systému definována proměnná prostředí SystemRoot. Tato proměnná prostředí je definována pouze na operačních systémech MS Windows od historického systému Windows NT až po nejnovější Windows 7.

A.2 První spuštění interaktivního rozhraní

Po úspěšné instalaci je nutné ověřit, zda je na daném počítači nainstalován inkrementální kompilátor GHCi, což ověříme příkazem ghci -- version, jehož výstupem je tento text:

*Tento příkaz je nutné spustit z adresáře _monalib_.

Samozřejmě se může lišit verze nainstalovaného kompilačního systému GHC [17].

Pro spuštění interaktivního rozhraní slouží příkaz make run*, který spustí GHCi a nahraje dynamickou knihovnu *monalib*.

Přímo z interaktivního rozhraní lze zadávat příkazy do shellu, z něhož je GHCi spuštěn. Pro tento účel slouží konstrukce :! *příkaz*. Praktické použití vypadá například takto:

```
*MONA> :! ls ..
Makefile        libmona.so        src
```

Důležitou vlastností, která je ze strany GHCi podporována, je našeptávání/doplňování aktuálně definovaných výrazů (tj. výrazů „pojmenovaných“ pomocí klauzule let a funkcí) a cest k souborům pomocí klávesy Tab, přičemž cesty k souborům jsou doplňovány při psaní řetězců relativně k aktuálnímu pracovnímu adresáři, tj. _monalib_/src. Tato vlastnost samozřejmě musí být podporována shellem, pod kterým je interaktivní rozhraní spouštěno.

Základní přehled příkazů, jenž poskytuje GHCi, je dostupné po zadání příkazu :?. Pro ukončení GHCi slouží příkaz :q.

A.3 Spuštění testů

Pro ověření implementace je připraveno několik sad základních testů, pomocí kterých jsou prověřeny klíčové funkce knihovny *monalib*.

Pro spuštění slouží funkce checkAll, která je definována v rámci modulu MONA_test.hs. Po spuštění testů je zobrazován následující výstup:

```
*MONA> checkAll
[[ Test 1 ]] -- Term construction & Conversion from term to string
+++ OK, passed 100 tests.
[[ Test 2 ]] -- Basic finite tree automata (prepared accepted terms)
@@@ OK, passed 6 tests.
[[ Test 3 ]] -- Basic finite tree automata (prepared rejected terms)
(160 tests)
```

Předchozí řádky zachycují proces testování ve stavu, kdy jsou úspěšně provedeny dvě sady testů. První sada se skládala ze 100 testů, jejichž vstupy byly automaticky generovány knihovnou QuickCheck, což lze na první pohled poznat podle toho, že řádek s hlášením o průběhu testů začíná prefixem +++. Druhá sada testů je založena na předpřipravených vstupech, což je indikováno tím, že řádek popisující výsledky dané sady testů začíná prefixem @@@. V první sadě bylo úspěšně provedeno 100 testů, kdežto v druhé sadě jich bylo provedeno pouze 6. Právě probíhá testování na třetí sadě testů, přičemž prozatím bylo úspěšně provedeno 160 testů.

Pro výpis stavu aktuálně počtu provedených testů je využíváno několik funkcí z modulu Test.QuickCheck.Text. Díky těmto funkcím je aktuální počet provedených testů přepisován stále na jednom řádku, což výrazným způsobem zjednodušuje orientaci v rámci výsledků již provedených testů. Nicméně knihovna QuickCheck je stále aktivně vyvíjena, a tak použité řešení nemusí být zcela kompatibilní s nejnovější verzí této knihovny. Aktuální implementace se opírá o knihovnu QuickCheck ve verzi 2.1.1.1, která je stále dostupná v rámci kompilačního systému GHC ve verzi 6.12. Ovšem pro novější verzi knihovny QuickCheck (konkrétně pro verzi 2.4.1.1) bude nutné udělat jeden zásah do zdrojového textu modulu MONA_test.hs,

*Tento příkaz je nutné spustit z adresáře _monalib_.

který se týká definice výrazu term. Tento zásah je doplněn patřičným komentářem, který je zároveň publikován v programové dokumentaci umístěné v adresáři Haskell_API.

Syntaxe volání funkce	**Popis**
importFTA $\mathcal{S}_M$ $\mathcal{S}_\Sigma$	Načte konečný stromový automat z definice uložené v souboru $\mathcal{S}_M$, který je definován nad abecedou definovanou v rámci souboru $\mathcal{S}_\Sigma$.
importTransducer $\mathcal{S}_\tau$ $\mathcal{S}_\Sigma$ $\mathcal{S}_{\Sigma'}$	Ze souboru $\mathcal{S}_\tau$ načte stromový převodník, jehož vstupní, resp. výstupní, abeceda je uložena v souboru $\mathcal{S}_\Sigma$, resp. $\mathcal{S}_{\Sigma'}$.
exportFTA $\mathcal{A}$ $\mathcal{S}_M$ $\mathcal{S}_\Sigma$	Uloží konečný stromový automat $\mathcal{A}$ do souboru $\mathcal{S}_M$. Pokud je řetězec $\mathcal{S}_\Sigma$ různý od prázdného řetězce, pak je do tohoto souboru uložena také abeceda, nad níž je definován $\mathcal{A}$. Očekává na standardním vstupu název automatu. Pokud není zadán použije se aktuální časová značka. Přepisuje existujících soubory!
exportFTA' $\mathcal{A}$ $\mathcal{S}_M$ $\mathcal{S}_\Sigma$	Bezpečnější verze předchozí funkce. Pokud již existuje některý ze souborů, do kterých mají být uloženy definice, pak je nutné potvrzení od uživatele, než dojde k přepsání původního souboru, resp. původních souborů.
printFTA $\mathcal{A}$	Vytiskne definici daného konečného stromového automatu na standardní výstup.
mona_printFTA $\mathcal{A}$	Na standardní výstup vytiskne konečný stromový automat pomocí funkcí GTA knihovny nástroje MONA, tzn. výpis zachycuje strukturu, jakou je daný automat reprezentován uvnitř nástroje MONA.
printTransducer $\mathcal{T}$	Na standardní výstup vytiskne definici daného stromového převodníku.
disposeFTA $\mathcal{A}$	Uvolní konečný stromový automat $\mathcal{A}$ z paměti.
disposeTransducer $\mathcal{T}$	Provede uvolnění paměti po stromovém převodníku $\mathcal{T}$.

Tabulka A.1: Přehled vstupních a výstupních funkcí

A.4 Praktické použití

Komentovaný přehled funkcí, které poskytuje interaktivní rozhraní pro knihovnu *mona-lib*, je rozdělen do dvojice tabulek. Nejprve je uvedena tabulka A.1, která znázorňuje přehled funkcí určených pro vstup a výstup konečných stromových automatů a stromových převodníků. Následuje tabulka A.2 zabývající se operacemi nad konečnými stromovými automaty, které jsou implementovány nad knihovnami nástroje MONA.

V obou těchto tabulkách je kvůli zkrácení zápisu syntaxe využito literálů, které mohou být pro odlišení opatřeny spodním indexem. Literál $\mathcal{S}$ označuje řetězec, který reprezentuje buď název souboru (může být uveden včetně cesty) nebo textovou formu termu. Pro reprezentaci konečného stromového automatu je určen literál $\mathcal{A}$. Pomocí literálu $\mathcal{T}$ je vyznačen argument reprezentující stromový převodník.

Jako první z příkladů použití je zde uvedeno načtení konečného stromového automatu, jeho výpis na standardní obrazovku a následné uvolnění paměti:

```
*MONA> let justA = importFTA "tests/basic/A_a.fta" "tests/basic/a-f.abc"
*MONA> justA
Warning: fta-lex/importFTA: State '__INIT__' is already in Q (ignored)
Reference: TAutomaton (on address #35456408)
*MONA> printFTA justA
============================
Alphabet
            a(001)
            b(010)
            c(100)
            d(110)
            e(101)
            f(011)
States
            __INIT__
            __TRAP__
            q_f
Final States
            q_f
Transitions
            a -> q_f
============================
*MONA> disposeFTA justA
Reference: No TAutomaton structure held
*MONA> printFTA justA
error: Cannot print 'NoAutomaton'!
*MONA>
```

Nejprve je ze zadaných souborů vytvořena struktura reprezentující konečný stromový automat, která je „pojmenována“ justA. Jelikož je Haskell založen na *lazy evaluation* vyhodnocovací strategii, je import struktury odložen až do první chvíle, kdy je tato struktura skutečně požadována, proto je na dalším řádku požádáno o vyhodnocení výrazu justA, při kterém je struktura načtena. Je nutné podotknout, že tento krok není nutné provádět. Struktura by se skutečně načetla při jejím prvním použití a nezáleží na kontextu použití – v tomto případě by byla struktura načtena právě před zavoláním funkce printFTA. V reakci na načtení struktury jsou na standardní výstup vytisknuty dva řádky. Na prvním řádku se nachází varování, které uživatele informuje o tom, že v definičním souboru konečného stromového automatu explicitně deklaroval stav __INIT__který je definován implicitně při importu nového automatu. Druhý řádek informuje uživatele o úspěšném načtení struktury včetně adresy, na které je struktura v paměti uložena. V případě neúspěšného načtení struktury by byl uživatel informován řádkem `Reference: No TAutomaton structure held`, kterému by předcházelo chybové hlášení. Můžeme si všimnout, že justA přijímá pouze jeden jediný základní term a tím je `a`.

Syntaxe funkce	**Typ výrazu**	**Popis**
union $\mathcal{A}_1$ $\mathcal{A}_2$	tAutomaton	Provede sjednocení zadané dvojice konečných stromových automatů. Navrací konečný stromový automat reprezentující jazyk $\mathrm{L}(\mathcal{A}_1) \cup \mathrm{L}(\mathcal{A}_2)$.
intersect $\mathcal{A}_1$ $\mathcal{A}_2$	tAutomaton	Provede průnik zadané dvojice konečných stromových automatů. Navrací konečný stromový automat reprezentující jazyk $\mathrm{L}(\mathcal{A}_1) \cap \mathrm{L}(\mathcal{A}_2)$.
complement $\mathcal{A}$	tAutomaton	Provede komplement zadaného konečného stromového automatu. Navrací konečný stromový automat reprezentující jazyk $\overline{\mathrm{L}(\mathcal{A})}$.
minimize $\mathcal{A}$	tAutomaton	Provede minimalizaci zadaného konečného stromového automatu. Navrací konečný stromový automat, který vznikne použitím funkce gtaMinimize z GTA knihovny nástroje MONA.
runTerm $\mathcal{A}$ $\mathcal{S}$	IO Bool	Zadaný konečný stromový automat použije pro simulaci běhu nad daným základním termem. Term je předáván ve formě řetězce. Jako návratová hodnota je vrácena hodnota Bool True, pokud byl běh přijímající, tj. $\mathrm{t_S} \in \mathrm{L}(\mathcal{A})$. V opačném případě je vrácena hodnota IO False.
runTerm' $\mathcal{A}$ $\mathcal{S}$	IO ()	„Upovídanější“ verze předchozí funkce. Na standardní výstup je vytisknut zadaný term, běh (binární strom ohodnocený stavy zadaného automatu) a informace o tom, zdali byl běh přijímající.
empty $\mathcal{A}$	IO Bool	Predikát prázdnosti jazyka reprezentovaného zadaným konečným stromovým automatem. Vrací IO True, platí-li $\mathrm{L}(\mathcal{A}) = \emptyset$. Jinak vrací IO False.
subset $\mathcal{A}_1$ $\mathcal{A}_2$	IO Bool	Predikát, který vrací IO True tehdy a jen tehdy, když platí vztah $\mathrm{L}(\mathcal{A}_1) \subseteq \mathrm{L}(\mathcal{A}_2)$. Jinak vrací IO False.
apply $\mathcal{T}$ $\mathcal{A}$	tAutomaton	Vrací konečný stromový automat reprezentující jazyk $\Upsilon_{\mathcal{T}}(\mathrm{L}(\mathcal{A}))$.

Tabulka A.2: Přehled poskytovaných funcí pro práci s konečnými stromovými automaty a stromovými převodníky

Poté je zavolána funkce printFTA, která vypíše definici konečného stromového automatu na standardní výstup. Výpis definice je z obou stran graficky ohraničen řádkem obsahujícím pouze znak =. Po výpisu definice konečného stromového automatu je pomocí funkce disposeFTA uvolněna jeho struktura z paměti, o čemž je uživatel informován na následujícím řádku. Následuje pokus o výpis definice struktury, která byla předtím uvolněna z paměti. Namísto výpisu definice je uživatel informován o tom, že se snaží vypsat struktury, kterou se nepodařilo úspěšně načíst nebo již byla uvolněna.

Následující příklad ilustruje použití funkcí union a runTerm'. Předpokládejme, že již máme z patřičných souborů načtenu dvojici konečných stromových automatů justA a justB.

```
*MONA> union justA justB
Reference: TAutomaton (on address #35519160)
*MONA> let u = it
*MONA> u
Reference: TAutomaton (on address #35519160)
*MONA> let t = "a(b, a)"
*MONA> runTerm' u t
Input term:
   a(b, a)
Run:
   __TRAP__(q_0, q_0)
Is run accepting:
   False
```

V tomto příkladu je znázorněna práce roztržitého uživatele, který si zapomenul pojmenovat výslednou strukturu reprezentující konečný stromový automat vzniklý použitím funkce union na danou dvojici parametrů. Řešením tohoto případu je jednoduché. GHCi automaticky spravuje referenci na poslední vyhodnocený výraz, který je dostupný přes „proměnnou“ it. V tomto případě stačí, aby uživatel ihned po příkazu použil klauzuli let pro zachycení reference, kterou uchovává it.

Následně je pomocí klauzule let „pojmenován“ řetězec, který je textovou reprezentací základního termu nad abecedou $\{\mathsf{a}, \mathsf{b}\}$. Poté je pomocí funkce runTerm' vyhodnocen běh konečného stromového automatu u nad základním termem t. Tato funkce tiskne na standardní výstup tyto informace:

- Za řádkem Input term: je vypsána textová reprezentace termu, nad kterým byl proveden běh konečného stromového automatu. V tomto případě je to term a(b, a).
- Po řádku Run: je vytisknut běh, což je v podstatě term, který má ohodnocen stavy konečného stromového automatu namísto symbolů z abecedy nad níž byl původní základní term definován.
- Nejdůležitějším je výsledek běhu, který je tisknut ihned za řádkem Is run accepting:. V tomto příkladu je běh konečného stromového automatu u nad základním termem t nepřijímající, tzn. term u nepatří do regulárního stromového jazyka daného konečného stromového automatu.

Adresářová struktura 1 Obsah přiloženého CD

- /
  - _monalib_
    - doc
      - C_API
        - ..... Dokumentace ve formátu HTML (vstupní bod index.html)
      - Haskell_API
        - ..... Dokumentace ve formátu HTML (vstupní bod index.html)
      - manual.pdf
    - src
      - test
        - ..... Testovací vstupy
      - alphabet.c
      - alphabet-lex.c
      - error.c
      - fta.c
      - fta-lex.c
      - lex.c
      - lib.c
      - symbol.c
      - term.c
      - alphabet.h
      - alphabet-lex.h
      - error.h
      - fta.h
      - fta-lex.h
      - lex.h
      - lib.h
      - return.h
      - symbol.h
      - term.h
      - MONA.hs
      - MONA_.hs
      - MONA_const.hs
      - MONA_FFI.hs
      - MONA_gen.hs
      - MONA_parse.hs
      - MONA_test.hs
      - MONA_const.hsc
      - Doxygen
      - Makefile
      - Makefile.win
    - Makefile
  - text
    - src
      - ..... Zdrojové texty této práce pro systém LaTeX
    - thesis.pdf
    - thesis-print.pdf